Programming with Unicorn

Very high level object-oriented application and system programming

Clinton Jeffery  Shamim Mohamed
Jafar Al Gharabaibeh Ray Pereda Robert Parlett
This page intentionally left blank.
Programming with Unicon

2nd edition

Clinton Jeffery
Shamim Mohamed
Jafar Al Gharraibeh
Ray Pereda
Robert Parlett
## Contents

Preface to the Second Edition  

### I  Core Unicon

#### 1  Programs and Expressions

1.1 Your First Unicon Program  
1.2 Command Line Options  
1.3 Expressions and Types  
1.4 Numeric Computation  
1.5 Strings and Csets  
1.6 Goal-directed Evaluation  
1.7 Fallible Expressions  
1.8 Generators  
1.9 Iteration and Control Structures  
1.10 Procedures  

#### 2  Structures

2.1 Tables  
2.2 Lists  
2.3 Records  
2.4 Sets  
2.5 Using Structures  
2.6 Summary  

#### 3  String Processing

3.1 String Indexes  
3.2 Character Sets  
3.3 Character Escapes  
3.4 String Scanning  
3.5 Regular Expressions  
3.6 Grammars
## 4 Advanced Language Features
- 4.1 Limiting or Negating an Expression ............................................. 53
- 4.2 List Structures and Parameter Lists ............................................. 54
- 4.3 Co-expressions ........................................................................ 55
- 4.4 User-Defined Control Structures ................................................. 56
- 4.5 Parallel Evaluation .................................................................. 57
- 4.6 Coroutines .............................................................................. 58
- 4.7 Permutations .......................................................................... 59
- 4.8 Simulation ................................................................................ 61

## 5 The System Interface
- 5.1 The Role of the System Interface ............................................... 65
- 5.2 Files and Directories .................................................................. 66
- 5.3 Programs and Process Control .................................................... 69
- 5.4 Networking .............................................................................. 73
- 5.5 Messaging Facilities .................................................................. 77
- 5.6 Tasks ....................................................................................... 79
- 5.7 Summary .................................................................................. 85

## 6 Databases
- 6.1 Language Support for Databases ................................................. 87
- 6.2 Memory-based Databases ............................................................. 88
- 6.3 DBM Databases ........................................................................ 88
- 6.4 SQL Databases ......................................................................... 90
- 6.5 Tips and Tricks for SQL Database Applications ............................. 96
- 6.6 Summary .................................................................................. 98

## 7 Graphics
- 7.1 2D Graphics Basics ................................................................ 99
- 7.2 Graphics Contexts ................................................................... 102
- 7.3 Events ..................................................................................... 104
- 7.4 Colors and Fonts ..................................................................... 106
- 7.5 Images, Palettes, and Patterns .................................................. 107
- 7.6 3D Graphics ........................................................................... 112
- 7.7 Textures .................................................................................. 118
- 7.8 Summary ................................................................................ 130

## 8 Threads
- 8.1 Threads and Co-Expressions ...................................................... 132
- 8.2 First Look at Unicon Threads ...................................................... 132
- 8.3 Thread Synchronization ............................................................. 136
13.3 Collaboration Diagrams ................................................. 223
13.4 Summary ................................................................. 224

III Example Applications .................................................. 225

14 CGI Scripts ................................................................. 227
  14.1 Introduction to CGI .................................................. 227
  14.2 The CGI Execution Environment .................................. 230
  14.3 An Example HTML Form .......................................... 231
  14.4 An Example CGI Script: Echoing the User’s Input ............ 233
  14.5 Debugging CGI Programs .......................................... 234
  14.6 Appform: An Online Scholarship Application ................. 234

15 System and Administration Tools .................................... 237
  15.1 Searching for Files .................................................. 237
  15.2 Finding Duplicate Files ............................................ 239
  15.3 User File Quotas ................................................... 245
  15.4 Capturing a Shell Command Session ............................ 250
  15.5 Filesystem Backups ................................................ 252
  15.6 Filtering Email ..................................................... 256
  15.7 Summary ............................................................. 261

16 Internet Programs .......................................................... 263
  16.1 The Client-Server Model .......................................... 263
  16.2 An Internet Scorecard Server .................................... 264
  16.3 A Simple “Talk” Program .......................................... 267
  16.4 Summary ............................................................. 273

17 Genetic Algorithms ....................................................... 275
  17.1 What are Genetic Algorithms? .................................... 275
  17.2 Operations: Fitness, Crossover, and Mutation .................. 276
  17.3 The GA Process ..................................................... 279
  17.4 ga_eng: a Genetic Algorithm Engine ............................ 280
  17.5 Color Breeder: a GA Application ................................ 285
  17.6 Picking Colors for Text Displays ............................... 287

18 Object-oriented User Interfaces ....................................... 289
  18.1 A Simple Dialog Example ......................................... 289
  18.2 Positioning Objects ................................................ 291
  18.3 A More Complex Dialog Example ............................... 292
  18.4 More About Event Handling ..................................... 298
Preface to the Second Edition

This book will raise your level of skill at computer programming, regardless of whether you are presently a novice or expert. The field of programming languages is still in its infancy, and dramatic advances will be made every decade or two until mankind has had enough time to think about the problems and principles that go into this exciting area of computing. The Unicon language described in this book is such an advance, incorporating many elegant ideas not yet found in most contemporary languages.

Unicon is an object-oriented, goal-directed programming language based on the Icon programming language. Unicon can be pronounced however you wish; we pronounce it variably depending on mood, whim, or situation; most frequently rhymes with “lexicon”.

For Icon programmers this work serves as a “companion book” that documents material such as the Icon Program Library, a valuable resource that is underutilized. Don’t be surprised by language changes: the book presents many new facilities that were added to Icon to make Unicon and gives examples from new application areas to which Unicon is well suited. For people new to Icon and Unicon, this book is an exciting guide to a powerful language.

It is with sweet irony that we call this book the 2nd Edition, since the first edition was never formally published but instead existed solely as an online document, although laser-printed hard copies could be requested. A lot has happened to Unicon since the first edition of this book, which culminated in 2004. This “2nd Edition” catches readers up with things like concurrent threads and vastly improved 3D graphics facilities. Along the way, the games chapter and parts of the internet programming chapter got spun off into a separate work, the so-called Manual of Puissant Skill at Game Programming.

Organization of This Book

This book consists of four parts. The first part, Chapters 1-8, presents the core of the Unicon language, much of which comes from Icon. These early chapters start with simple expressions, progress through data structures and string processing, and include advanced programming topics and the input/output capabilities of Unicon’s portable system interface. Part two, in Chapters 9-12, describes object-oriented development as a whole and presents Unicon’s object-oriented facilities in the context of object-oriented design. Object-
oriented programming in Unicon corresponds closely to object-oriented design diagrams in
the Unified Modeling Language, UML. Some of the most interesting parts of the book are
in part three; Chapters 13-18 provide example programs that use Unicon in a wide range
of application areas. Part four consists of essential reference material presented in several
Appendixes.

Acknowledgments

Thanks to the Icon Project for creating a most excellent language. Thanks especially to
those unsung heroes, the university students and Internet volunteers who implemented the
language and its program library over a period of many years. Icon contributors can be
divided into epochs. In the epoch leading up to the first edition of this book, we were
inspired by contributions from Gregg Townsend, Darren Merrill, Mary Cameron, Jon Lipp,
Anthony Jones, Richard Hatch, Federico Balbi, Todd Proebsting, Steve Lumos and Naomi
Martinez. In the epoch since the first edition of this book, the Unicon Project owes a debt
of gratitude to Ziad al Sharif, Hani bani Salameh, Jafar Al Gharaibeh, Mike Wilder, and
Sudarshan Gaikaiwari.

The most impressive contributors are those whose influence on Icon has spanned across
epochs, such as Ralph Griswold, Steve Wampler, Bob Alexander, Ken Walker, Phillip
Thomas, and Kostas Oikonomou. We revere you folks! Steve Wampler deserves extra
thanks for serving as the technical reviewer for the first edition of this book. Phillip
Thomas and Kostas Oikonomou have provided extensive support and assistance that goes
way beyond the call of duty; in many ways this is their book.

This manuscript received critical improvements and corrections from many additional
technical reviewers, including, David A. Gamey, Craig S. Kaplan, David Feustel, David
Slate, Frank Lhota, Art Eschenlauer, Wendell Turner, Dennis Darland, and Nolan Clayton.

The authors wish to acknowledge generous support from the National Library of
Medicine and AT&T Bell Labs Research. This work was also supported in part by the
National Science Foundation under grants CDA-9633299, EIA-0220590 and EIA-9810732,
and the Alliance for Minority Participation.

Clinton Jeffery
Shamim Mohamed
Jafar al Gharaibeh
Ray Pereda
Robert Parlett
Introduction

Software development requires thinking about several dimensions simultaneously. For large programs, writing the actual computer instructions is not as difficult as figuring out the details of what the computer is supposed to do. After analyzing what is needed, program design brings together the data structures, algorithms, objects, and interactions that accomplish the required tasks. Despite the importance of analysis and design, programming is still the central act of software development for several reasons. The weak form of the Sapir-Whorf hypothesis suggests that the programming language we use steers and guides the way we think about software, so it affects our designs. Software designs are mathematical theorems, while programs are proofs that test those designs. As in other branches of mathematics, the proofs reign supreme. In addition, a correct design can be foiled by an inferior implementation.

This book is a guide and reference for an exciting programming language called Unicon that has something to offer both computer scientists as well as casual programmers. You will find explanations of fundamental principles, unique language idioms, and advanced concepts and examples. Unicon exists within the broader context of software development, so the book also covers software engineering fundamentals. Writing a correct, working program is the central task of software engineering. This does not happen automatically as a result of the software design process. Make no mistake: if you program very much, the programming language you use is of vital importance. If it weren’t, we would still be programming in machine language.

Prototyping and the Spiral Model of Development

A software prototype is a working subset of a software system. Prototypes help check software designs and user interfaces, demonstrate key features to customers, or prove the feasibility of a proposed solution. A prototype may generate customer feedback on missing functionality, provide insight on how to improve the design, lead to a decision about whether to go ahead with a project or not, or form a starting point for the algorithms and data structures that will go into the final product. Prototyping is done early in the software development process. It fits naturally into the spiral model of development proposed by Barry Boehm (1988). Figure I-1 shows the spiral model; time is measured by the distance
from the center. Analysis, design, coding, and evaluation are repeated to produce a better product with each iteration. "Prototyping" is the act of coding during those iterations when the software is not yet fully specified or the program does not yet remotely implement the required functionality.

![The Spiral Model of Software Development](image)

Tight spirals are better than loose spirals. The more powerful the prototyping tools, the less time and money spent in early iterations of development. This translates into either faster time to market, or a higher quality product. Some prototypes are thrown away once they have served the purpose of clarifying requirements or demonstrating some technique. This is OK, but in the spiral model some prototypes are gradually enhanced until they become the final production system.

**Icon: a Very High Level Language for Applications**

Icon is a programming language developed at the University of Arizona. Icon generalizes its developers’ experience creating an earlier language, SNOBOL4. Icon embodies seminal research ideas, but it is also more fun and easier to program than other languages. Most very high-level languages revel in cryptic syntax, while Icon is not just more powerful, but often more readable than its competitors. This gain in expressive power without losing readability is an addicting result of Icon’s elegant design.

The current Arizona Icon, version 9.5, is described in *The Icon Programming Language, 3rd edition* by Ralph and Madge Griswold (1996). Its reference implementation is a virtual machine interpreter. Icon evolved through many releases over two decades and is far more capable than it was originally. It is apparently a finished work.

**Enter Unicon: More Icon than Icon**

The name “Unicon” refers to the descendant of Icon described in this book and distributed from [www.unicon.org](http://www.unicon.org). Unicon is Icon with portable, platform-independent access to hard-
ware and software features that have become ubiquitous in modern applications development, such as objects, networks, and databases. Unicon is created from the same public domain source code that Arizona Icon uses, so it has a high degree of compatibility. We were not free to call it version 10 of the Icon language, since it was not produced or endorsed by the Icon Project at the University of Arizona.

Just as the name Unicon frees the Icon Project of all responsibility for our efforts, it frees us from the requirement of backward compatibility. While Unicon is almost entirely backward compatible with Icon, dropping full compatibility allows us to clear out some dead wood and more importantly, to make some improvements in the operators that will benefit everyone at the expense of...no one but the compatibility police. This book covers the features of Icon and Unicon together. A compatibility check list and description of the differences between Icon and Unicon are given in Appendix D.

The Programming Languages Food Chain

It is interesting to compare Icon and Unicon with the competition. Mainstream programming languages such as C, C++, and Java, like the assembler languages that were mainstream before them, are ideal tools for writing all sorts of programs, so long as vast amounts of programmer time are available. Throwing more programmers at a big project does not work well, and programmers are getting more expensive while computing resources continue to become cheaper. These pressures inexorably lead to the use of higher-level languages and the development of better design and development methods. Such human changes are incredibly slow compared to technological changes, but they are visibly occurring nevertheless. Today, the most productive programmers are using extra CPU cycles and memory to reduce the time it takes to develop useful programs.

There is a subcategory of mainstream languages, marketed as rapid application development languages, whose stated goals seem to address this phenomenon. Languages such as Visual Basic or PowerBuilder provide graphical interface builders and integrated database connectivity, giving productivity increases in the domain of data entry and presentation. The value added in these products are in their programming environments, not their languages. The integrated development environments and tools provided with these languages are to be acclaimed and emulated, but they do not provide productivity gains that are equally relevant to all application domains. They are only a partial solution to the needs of complex applications.

Icon is designed to be easier and faster to program than mainstream languages. The value it adds is in the expressive power of the language itself, in the category of very high level languages that includes Lisp, APL, Smalltalk, REXX, Perl, Tcl, Python, and Ruby; there are many others. Very high-level languages can be subdivided into scripting languages and applications languages. Scripting languages often glue programs together from disparate sources. They are typically strong in areas such as multilingual interfacing and file
system interactions, while suffering from weaker expression semantics, typing, scope rules, and control structures than their applications-oriented cousins. Applications languages typically originate within a particular application domain and support that domain with special syntax, control structures, and data types. Since scripting is an application domain, scripting languages are just one prominent subcategory of very high-level languages.

Icon is an applications language with roots in text processing and linguistics. Icon programs tend to be more readable than similar programs written in other very high-level languages, making Icon well-suited to the aims of literate programming. For example, Icon was used to implement Norman Ramsey’s literate programming tool noweb (Ramsey, 1994). Literate programming is the practice of writing programs and their supporting textual description together in a single document.

Unicon makes the core contributions of Icon useful for a broader range of applications. This book’s many examples illustrate the range of tasks for which Unicon is well suited, and these examples are the evidence in support of Unicon’s existence. Consider using Unicon when one or more of the following conditions are true. The more conditions that are true, the more you will benefit from Unicon.

- Programmer time must be minimized.
- Maintainable, concise source code is desired.
- The program includes complex data structures or experimental algorithms.
- The program involves a mixture of text processing and analysis, custom graphics, data manipulation, network or file system operations.
- The program must run on several operating systems and have a nearly identical graphical user interface with little or no source code differences.

Unicon is not the last word in programming. You probably should not use Unicon if your program has one or more of the following requirements:

- The fastest possible performance is needed.
- The program has hard real-time constraints.
- The program must perform low-level or platform-specific interactions with the hardware or operating system.

Programming languages play a key role in software development. The Unicon language is a very high level object-oriented language with a unique combination of expressive power and scalable rapid development. In this book, many examples from a wide range of application areas demonstrate how to apply and combine Unicon’s language constructs to solve real-world problems. It is time to move past the introductions. Prepare to be spoiled by this language. You may have the same feelings that Europeans felt when they gave up using Roman numerals and switched to the Hindu-Arabic number system. “This multiplication stuff isn’t that hard anymore!”
Part I

Core Unicon
Chapter 1

Programs and Expressions

This chapter presents many of the key features of Unicon, starting with those it has in common with other popular languages. Detailed instructions show how to compile and run programs. Soon the examples introduce important ways in which Unicon is different from other languages. These differences are more than skin deep. If you dig deeply, you can find dozens of details where Unicon provides just the right blend of simplicity, flexibility, and power. After this chapter, you will know how to

- edit, compile, and execute Unicon programs
- use the basic types to perform calculations
- identify expressions that can fail, or produce multiple results
- control the flow of execution using conditionals, looping, and procedures

1.1 Your First Unicon Program

This section presents the nuts and bolts of writing and running an Unicon program, after which you will be able to try the code examples or write your own programs. Before you can run the examples here or in any subsequent chapter, you must install Unicon on your system. (See Appendix F for details on downloading and installing Unicon from the Unicon web site, http://unicon.org.) We are going to be very explicit here, and assume nothing about your background. If you are an experienced programmer, you will want to skim this section, and move on to the next section. If you are completely new to programming, have no fear. Unicon is pretty easy to learn.

All programs consist of commands that use hardware to obtain or present information to users, and perform calculations that transform information into a more useful form. To program a computer you write a document containing instructions for the computer to carry out. In Unicon a list of instructions is called a procedure, and a program is a collection of one or more procedures. In larger programs, groups of related procedures are organized
CHAPTER 1. PROGRAMS AND EXPRESSIONS

into classes or packages; these features are presented in Part II of this book. Unicon programs are text files that may be composed using any text editor. For the purposes of demonstration this section describes how to use Ui, the program editor and integrated development tool that comes with Unicon.

It is time to begin. Fire up Ui by typing "ui" from the command line, or launching the menu item or icon labeled "Unicon," and type:

```unicon
procedure main()
    write("Hello, amigo!")
end
```

Your screen should look something like Figure 1-1. The large upper area of the window is the editing region where you type your program code. The lower area of the window is a status region in which the Ui program displays a message when a command completes successfully, or when your program has an error. Until you explicitly name your file something else, a new file has the name `noname.icn`. The font Ui uses to display source code is selectable from the Options menu.

Figure 1-1: Writing an Unicon program using the Ui program.
1.1. YOUR FIRST UNICON PROGRAM

The list of instructions that form a procedure begins with the word procedure and ends with the word end. Procedures have names. After writing a list of instructions in a procedure you may refer to it by name without writing out the list again. The write() instruction is just such a procedure, only it is already written for you; it is built in to the language. When you issue a write() instruction, you tell the computer what to write. The details a procedure uses in carrying out its instructions are given inside the parentheses following that procedure’s name; in this case, "Hello, amigo!" is to be written. When you see parentheses after a name in the middle of a list of instructions, it is an instruction to go execute that procedure’s instructions. Inside the parentheses there may be zero, one, or many values supplied to that procedure.

Besides writing your program, there are a lot of menu commands that you can use to control the details of compiling and executing your program within Ui. For instance, if you select Run→Run, Ui will do the following things for you.

1. Save the program in a file on disk. All Unicon programs end in .icn; you could name it anything you wished, using the File→SaveAs command.

2. Compile the Unicon program from human-readable text to (virtual) machine language. To do this step manually, you can select the Compile→Make executable command.

3. Execute the program. This is the main purpose of the Run command. Ui performed the other steps in order to make this operation possible.

If you type the hello.icn file correctly, the computer should chug and grind its teeth for awhile, and

Hello, amigo!

should appear in a window on your screen. This ought to be pretty intuitive, since the instructions included the line

write("Hello, amigo!")

in it. That’s how to write to the screen. It’s that simple.

The first procedure to be executed when a program runs is called main(). Every instruction listed in the procedure named main() is executed in order, from top to bottom, after which the program terminates. Use the editor to add the following lines right after the line write("Hello, amigo") in the previous program:

write("How are you?")
write(7 + 12)

The end result after making your changes should look like this:
procedure main()
  write("Hello, amigo!")
  write("How are you?")
  write(7 + 12)
end

Run the program again. This example shows you what a list of instructions looks like, as well as how easy it is to tell the computer to do some arithmetic.

Note
It would be fine (but not very useful) to tell the computer to add 7 and 12 without telling it to write the resulting value. On seeing the instruction

7 + 12

the computer would do the addition, throw the 19 away, and go on.

Add the following line, and run it:

write("7 + 12")

This illustrates what quotes are for. Quoted text is taken literally; without quotes, the computer tries to simplify (do some arithmetic, or compute the value of what is written), which might be difficult if the material in question is not an expression!

write(hey you)

makes no sense and is an error. Add this line, and run it:

write(7 + "12")

The 12 in quotes is taken literally as some text, but that text happens to be digits that comprise a number, so adding it to another number makes perfect sense. The computer will not have as much success if you ask it to add 7 to “amigo”. The computer views all of this in terms of values. A value is a unit of information, such as a number. Anything enclosed in quotes is a single value. The procedure named write() prints values on your screen. Operators such as + take values and combine them to produce other values, if it is possible to do so. The values you give to + had better be numbers! If you try to add something that doesn’t make sense, the program will stop running at that point, and print an error message.

By now you must have the impression that writing things on your screen is pretty easy. Reading the keyboard is just as easy, as illustrated by the following program:

procedure main()
  write("Type a line ending with <ENTER>:")
  write("The line you typed was", read())
end
Run the program to see what it does. The procedure named read() is used to get what the user types. It is built in to the language. The read() instruction needs no directions to do its business, so nothing is inside the parentheses. When the program is run, read() grabs a line from the keyboard, turns it into a value, and produces that value for use in the program, in this case for the enclosing write() instruction.

The write() instruction is happy to print out more than one value on a line, separated by commas. When you run the program, the "the line you typed was" part does not get printed until after you type the line for read() and that instruction completes. The write() instruction must have all of its directions (the values inside the parentheses) before it can go about its business.

Now let’s try some variations. Can you guess what the following line will print?

write("this line says ", "read()")

The read() procedure is never executed because it is quoted! Quotes say "take these letters literally, not as an equation or instruction to evaluate.” How about:

write("this line says , read()")

Here the quotes enclose one big value, which is printed, comma and all. The directions one gives to a procedure are parameters; when you give a procedure more than one parameter, separated by commas, you are giving it a parameter list. For example,

write("this value ", "and this one")

Compile and run the following strange-looking program. What do you think it does?

procedure main()
    while write( "" ˜== read() )
end

This program copies the lines you type until you type an empty line by pressing Enter without typing any characters first. The "" are used just as usual. They direct the program to take whatever is quoted literally, and this time it means literally nothing - an empty line. The operator ˜== stands for "not equals”. It compares the value on its left to the value on its right, and if they are not equal, it produces the value on the right side; if they are equal, it fails - that is, the “not equals” operator produces no value. If you have programmed in other languages, this may seem like a strange way to describe what is usually performed with nice simple Boolean values True and False. For now, try to take this description at face value; Unicon has no Boolean type or integer equivalent, it uses a more powerful concept that we will examine more fully in the chapters that follow.

Thus, the whole expression "" ˜== read() takes a line from the keyboard, and if it is not empty, it produces that value for the enclosing write() instruction. When you type an empty
line, the value `read()` produces is equal to "", and `~==` produces no value for the enclosing `write()` instruction, which similarly fails when given no value. The `while` instruction is a “loop” that repeats the instruction that follows it until that instruction fails (in this case, until there is no more input). There are other kinds of loops, as well as another way to use `while`; they are all described later in this chapter.

So far we’ve painted you a picture of the Unicon language in very broad strokes, and informally introduced several relevant programming concepts along the way. These concepts are presented more thoroughly and in their proper contexts in the next sections and subsequent chapters. Hopefully you are already on your way to becoming an Icon programmer extraordinaire. Now it is time to dive into many of the nuts and bolts that make programming in Unicon a unique experience.

1.2 Command Line Options

Unicon comes with an IDE, but you can edit programs with any editor, and compile and run them from your operating system’s command line. This section describes the Unicon command line tools along with several useful options. The Unicon compiler executable is named `unicon`, and to compile the program `foo.icn` you would type

```
unicon foo
```

To execute the resulting program, just type

```
foo
```

To compile and link a program consisting of several modules, you can type them all on the command line, as in

```
unicon foo bar baz
```

but often you will want to compile them separately (using the `-c` command line option) and link the resulting object files, called ucode files; their extension is `.u`

```
unicon -c foo
unicon -c bar
unicon -c baz
unicon foo.u bar.u baz.u
```

Some of the other useful command line options include:

- `-o arg` name the resulting output file `arg`

- `-x args` execute the program immediately after linking; this option goes after the program filenames
1.3. EXPRESSIONS AND TYPES

- **-t** turn on tracing
- **-u** produce a warning for undeclared variables

These options can be specified in the Ui program under the Compile menu’s Compile Options command. Other options exist; consult your Unicon and Icon manual pages and platform-specific help files and release information for more details.

### 1.3 Expressions and Types

Each procedure in a Unicon program is a sequence of expressions. Expressions are instructions for obtaining values of some type, such as a number or a word; some expressions also cause side effects, such as sending data to a hardware device. Simple expressions just read or write a value stored in memory. More interesting expressions specify a computation that manipulates zero or more argument values to obtain result values by some combination of operators, procedures, or control structures.

The simplest expressions are literals, such as 2 or "hello, world!". These expressions directly specify a value stored in memory. When the program runs, they do not do any computation, but rather evaluate to themselves. Literals are combined with other values to produce interesting results. Each literal has a corresponding type. This chapter focuses on the atomic types. Atomic types represent individual, immutable values. The atomic types in Unicon are integer and real (floating-point) numbers, string (a sequence of characters), and cset (a character set). Atomic types are distinguished by the fact that they have literal values, specified directly in the program code and represented as data in the compiled code. Values of other types such as lists are constructed during execution. Later chapters describe structure types that organize collections of values, and system types for interacting with the operating system via files, databases, windows, and network connections.

After literals, references to variables are the next simplest form of expression. Variables are named memory locations that hold values for use in subsequent expressions. You refer to a variable by its name, which must start with a letter or underscore and may contain any number of letters, underscores, or numbers. Use names that make the meaning of the program clear. The values stored in variables are manipulated by using variable names in expressions like i+j. This expression results in a value that is the sum of the values in the variables i and j, just like you would expect.

Some words may not be used as variable names because they have a special meaning in the language. These reserved words include procedure, end, while, and so on. Other special variables called keywords start with the ampersand character (&) and denote special values. For example, global variables are initialized to the null value represented by the keyword &null. Other keywords include &date, &time, and so on. Complete lists of reserved words and keywords are given in Appendix A.
Unlike many languages where you have to state up front (declare) all the variables you are going to use and specify their data type, in Unicon variables do not have to be declared at all, and any variable can hold any type of value. However, Unicon will not allow you to mix incompatible types in an expression. Unicon is type safe, meaning that every operator checks its argument values to make sure they are compatible, converts them if necessary, and halts execution if they cannot be converted.

1.4 Numeric Computation

Unicon supports the usual arithmetic operators on data types integer and real. Integers are signed whole numbers of arbitrary magnitude. The real type is a signed floating point decimal number whose size on any platform is the largest size supported by machine instructions, typically 64-bit double precision values. In addition to addition (+), subtraction (-), multiplication (*) and division (/), there are operators for modulo (%) and exponentiation (^). Arithmetic operators require numeric operands.

Note

Operations on integers produce integers; fractions are truncated, so 8/3 produces 2. If either operand is a real, the other is converted to real and the result is real, so 8.0/3 is 2.66666...

As a general rule in Unicon, arguments to numeric operators and functions are automatically converted to numbers if possible, and a run-time error occurs otherwise. The built-in functions integer(x) and real(x) provide an explicit conversion mechanism that fails if x cannot be converted to numeric value, allowing a program to check values without resulting in a run-time error.

In addition to the operators, built-in functions support several common numeric operations. The sqrt(x) function produces the square root of x, and exp(x) raises e to the x power. The value of pi (3.141...) is available in keyword &pi, the Golden Ratio (1.618...) is available in &phi, and e (2.718...) is available in &e. The log(x) function produces the natural log of x. The common trigonometric functions, such as sin() and cos() take their angle arguments in radian units. The min(x1, x2, ...) and max(x1, x2, ...) routines return minimum and maximum values from any number of arguments.

Listing 1-1 shows a simple Unicon program that illustrates the use of variables in a numeric computation. The line at the beginning is a comment for the human reader. Comments begin with the # character and extend to the end of the line on which they appear. The compiler ignores them.

Listing 1-1 Mystery program

```plaintext
# What do I compute?
procedure main()
```
local i, j, old_r, r
i := read()
j := read()
old_r := r := min(i, j)
while r > 0 do {
    old_r := r
    if i > j then
        i := r := i % j
    else
        j := r := j % i
}
write(old_r)
end

This example illustrates assignment; values are assigned to (or "stored in") variables with the := operator. As you saw in the previous section, the function read() reads a line from the input and returns its value. The modulo operator (%) is an important part of this program: i % j is the remainder when i is divided by j.

While loops can use a reserved word do followed by an expression (often a compound expression in curly braces). The expression following the do is executed once each time the expression that controls the while succeeds. Inside the while loop, a conditional if-then-else expression is used to select from two possible actions.

The names of the variables in this example are obscure, and there are no comments in it other than the one at the top. Can you guess what this program does, without running it? If you give up, try running it with a few pairs of positive numbers.

In addition to arithmetic operators, there are augmented assignment operators. To increment the value in a variable by 2, these two statements are equivalent:

\[
i +:= 2
\]
\[
i := i + 2
\]

Augmented assignment works for most binary operators, not just arithmetic. The expression \(i \ op := expr\) means the same as \(i := i \ op \ expr\).

1.5 Strings and Csets

The non-numeric atomic types in Unicon are character sequences (strings) and character sets (csets). Icon came from the domain of string processing, and from it Unicon inherits many sophisticated features for manipulating strings and performing pattern matching. This section presents the simple and most common operations. More advanced operations and examples using strings and csets are given in Chapter 4.
String literals are enclosed in double quotes, as in "this is a string", while cset literals are enclosed in single quotes, as in 'aeiou'. Although strings and csets are composed of characters, there is no character type; a string (or cset) consisting of a single character is used instead.

Current implementations of Unicon use eight-bit characters, allowing strings and csets to be composed from 256 unique characters. ASCII representation is used for the lower 128 characters, except on EBCDIC systems. The appearance of non-ASCII values is platform dependent. Like integers, strings can be arbitrarily large, constrained only by the amount of memory you have on your system.

Several operators take string arguments. The *s operator gives the length of string s. The expression s1||s2 produces a string consisting of the characters in s1 followed by s2. The subscript operator s[i] produces a one-letter substring of s at the ith position. Indices are counted starting from position 1. If i is nonpositive, it is from the end of the string, for example s[-2] is the second to the last character in the string.

Csets support set operators. c1++c2 produces a cset that is the union of c1 and c2. The expression c1**c2 is the intersection, while c1--c2 is the difference. In addition, several keywords are commonly used csets. The keywords &letters, &lcase, and &ucase denote the alphabetic characters, lower case characters a-z, and upper case characters A-Z, respectively, while &digits is the set from 0-9, &ascii is the lower 128 characters, and &cset is the set of all (256, on most implementations) characters.

Many built-in functions operate on strings and csets. Some of the simple string functions are reverse(s), which produces a string that is the reverse of s, and trim(s,c), which produces a substring of s that does not end with any character in cset c.

Functions and operators that require string arguments convert numeric values to strings automatically, and halt execution with a run-time error if given a value that cannot be converted to a string.

1.6 Goal-directed Evaluation

So far, the examples of how expressions are evaluated have included nothing you wouldn’t find in ordinary programming languages. It is time to push past the ordinary. In most conventional languages, each expression always computes exactly one result. If no valid result is possible, a sentinel value such as -1, NULL, EOF (end-of-file) or INF (infinity) is returned instead. This means that the program must check the return value for this condition. For example, while reading integers from the input and performing some operation on them you might do something like this:

```unicon
while (i := read()) ^= -1 do
    process(i)
```
This will work, of course, except when you really need to use -1 as a value! It is somewhat cumbersome, however, even when a sentinel value is not a problem. Unicon provides a much nicer way to write this type of code, developed originally in the Icon language. In Unicon, expressions are goal-directed. This means that every expression when evaluated has a goal of producing results for the surrounding expression. If an expression succeeds in producing a result, the surrounding expression executes as intended, but if an expression cannot produce a result, it is said to fail and the surrounding expression cannot be performed and in turn fails.

Now take a look at that loop again. If it weren’t for the termination condition, you would not need the intermediate variable $i$. If you would like to say:

$$\text{process(read())}$$

your wishes are answered by Unicon; you can indeed write your program like this. The expression $\text{read()}$ tries to produce a value by reading the input. When it is successful, $\text{process()}$ is called with the value; but when $\text{read()}$ cannot get any more values, that is, at the end of the file, it fails. This failure propagates to the surrounding expression and $\text{process()}$ is not called either. Here is the clincher: control expressions like if and while don’t check for Boolean (true/false) values, they check for success! So our loop becomes

$$\text{while process(read())}$$

The do clause of a while loop is optional; in this case, the condition does everything we need, and no do clause is necessary.

Consider the if statement that was used in the earlier arithmetic example:

$$\text{if } i > j \text{ then } ...$$

Comparison operators such as $>$ succeed or fail depending on the values of the operands. This leads to another question: if an expression like $i < 3$ succeeds, what value should it produce? No “true” value is needed, because any result other than failure is interpreted as “true.” This allows the operator to return a useful value instead! The comparison operators produce the value of their right operand when they succeed. You can write conditions like

$$\text{if } 3 < i < 7 \text{ then } ...$$

that appear routinely in math classes. Other programming languages only dream about being this elegant. First, Unicon computes $3 < i$. If that is true, it returns the value $i$, which is now checked with $7$. This expression in fact does exactly what you’d expect. It checks to see that the value of $i$ is between 3 and 7. (Also, notice that if the first comparison fails, the second one will not be evaluated.)
1.7 Fallible Expressions

Because some expressions in Unicon can fail to produce a result, you should learn to recognize such expressions on sight. These fallible expressions control the flow of execution through any piece of Unicon code you see. When failure is expected it is elegant. When it is unexpected in your program code, it can be disastrous, causing incorrect output that you may not notice or, if you are lucky, the program may terminate with a run-time error.

Some fallible expressions fail when they cannot perform the required computation; others are predicates whose purpose is to fail if a condition is not satisfied. The subscript and sectioning operators are examples of the first category. The expression \( x[i] \) is a subscript operator that selects element \( i \) out of some string or structure \( x \). It fails if the index \( i \) is out of range. Similarly, the sectioning operator \( x[i:j] \) fails if either \( i \) or \( j \) are out of range.

The \( \text{read()} \) function is illustrative of a large number of built-in functions that can fail. A call to \( \text{read()} \) fails at the end of a file. You can easily write procedures that behave similarly, failing when they cannot perform the computation that is asked. Unfortunately, for an arbitrary procedure call \( p() \), you can’t tell if it is fallible without studying its source code or reference documentation. The safest thing is to expect any procedure call is fallible and check whether it failed, unless you know it is not fallible or its failure doesn’t matter. Following this advice may avoid many errors and save you lots of time. In this book we will be careful to point out fallible expressions when we introduce them.

The less than operator \( < \) is a typical predicate operator, one that either fails or produces exactly one result. The unary predicates /\( x \) and \( \backslash x \) test a single operand, succeeding and producing the operand if it is null, or non-null, respectively. The following binary predicates compare two operands. The next section presents some additional, more complex fallible expressions.

\[
\begin{align*}
< & \quad \leq & \quad > & \quad \geq & \quad = & \quad \neq & \\
\ll & \quad \ll & \quad \gg & \quad \ggg & \quad == & \quad \neq == & \\
& \quad \equiv & \quad \neq & \quad \equiv & 
\end{align*}
\]

numeric comparison operators

lexical (alphabetic) comparison

reference comparison

1.8 Generators

So far we have seen that an expression can produce no result (failure) or one result (success). In general, an expression can produce any number of results: 0, 1, or many. Expressions that can produce more than one result are called generators. Consider the task of searching for a substring within a string:

\[
\text{find("lu", "Honolulu")}
\]

In most languages, this would return one of the substring matches, usually the first position at which the substring is found. In Unicon, this expression is a generator, and
can produce all the positions where the substring occurs. If the surrounding expression
only needs one value, as in the case of an if test or an assignment, only the first value of a
generator is produced. If a generator is part of a more complex expression, then the return
values are produced in sequence until the whole expression produces a value.

Let us look at this example:

\[ 3 < \text{find("or", "horror")} \]

The first value produced by \text{find()} is 2, which causes the \text{<} operation to fail. Execution
then resumes the call to \text{find()}, which produces a 5 as its next value, and the expression
succeeds. The value of the expression is the first position of the substring greater than 3.

The most obvious generator is the alternation operator |. The expression

\[ \text{expr1} | \text{expr2} \]

produces its left-hand side followed by its right-hand side, if needed by the surrounding
expression. This can perform many computations quite compactly. For example,

\[ x = (3 | 5) \]

checks to see if the value of \( x \) is 3 or 5. More complex expressions follow logically:

\[ (x | y) = (3 | 5) \]

checks to see if either \( x \) or \( y \) has the value 3 or 5. It is the Unicon equivalent of C’s

\[ (x == 3) || (x == 5) || (y == 3) || (y == 5) \]

In understanding Unicon code, it helps if you identify the generators, if there are any.
In addition to the alternation operator | and the function \text{find()}, there are a few other
generators in Icon’s built in repertoire of operators and functions. We mention them briefly
here, so you can be on the lookout for them when reading code examples.

The expression \( i \) to \( j \) is a generator that produces all the values between \( i \) and \( j \). The expression \( i \) to \( j \) by \( k \) works similarly, incrementing each result by \( k \); \( i \), \( j \), and \( k \) must all be
integer or real numbers, and \( k \) must be non-zero. The expression \( i \) to \( j \) is equivalent to \( i \) to \( j \) by 1. The unary ! operator is a generator that produces the elements of its argument.
This works on every type where it makes sense. Applied to a string, it produces all its
characters (in order). Sets, tables, lists, or records produce the members of the structure.

Generators get resumed for more results as needed in order for the surrounding expres-
sion to succeed, and this may propagate through many levels of nested enclosing expressions.
However, special expressions called \textit{bounded expressions} will never resume their generator
subexpressions. For example, the conditional expressions used in \textit{if} and \textit{while} are never
resumed if they succeed; if they produce a result the then-branch or the loop body is exe-
cuted, and if that code fails, it does not cause generators in the conditional to be resumed.
Those bounded conditional expressions are re-evaluated starting from scratch if execution comes their way again. Another popular bounded expression is the semi-colon operator. The expression \( \text{expr}_1 \; ; \; \text{expr}_2 \) evaluates the two expressions in order, and bounds the first expression, so you don’t have to worry about backtracking into it if the second expression fails.

1.9 Iteration and Control Structures

You have already seen two control structures in Unicon: the if and the while loop, which test for success. Unicon has several other control structures that vary in complexity and usefulness. Unicon is expression-based, and all control structures are expressions that can be used in surrounding expressions if desired. The big difference between control structures and ordinary operators or procedures is that ordinary operators and procedures don’t execute until their arguments have been evaluated and produced a result; they don’t execute at all if an argument fails. In contrast, a control structure uses one of its arguments to decide whether (or how many times) to evaluate its other arguments.

Since control structures are expressions, they may produce a result for a surrounding expression. For example, the result of an if expression is the result of either its then part or its else part, whichever one was selected. On the other hand, a loop executes until its test fails, after which there is no meaningful result for it to produce; loops usually fail as far as surrounding expressions are concerned.

The control structure every processes the entire sequence of values produced by a generator. The expression

\[
every \, \text{expr}_1 \, \text{do} \, \text{expr}_2
\]

evaluates \( \text{expr}_2 \) for each result generated by \( \text{expr}_1 \). This loop looks similar enough to a while loop to confuse people at first. The difference is that a while loop re-evaluates \( \text{expr}_1 \) from scratch after each iteration of \( \text{expr}_2 \), but every resumes \( \text{expr}_1 \) for an additional result where it left off the last time through the loop. Using a generator to control a while loop makes no sense; the generator will restart each iteration, which may give you an infinite loop. Similarly, not using a generator to control an every loop also makes no sense; if \( \text{expr}_1 \) is not a generator the loop body executes at most one time.

The classic example of every is a loop that generates the number sequence from a to expression, assigning the number to a variable that can be used in \( \text{expr}_2 \). In many languages these are called “for” loops. A for loop in Unicon is written like this:

\[
every \, i \, := \, 1 \, \text{to} \, 10 \, \text{do} \, \text{write}(i)
\]

Of course, every and to are not limited to this BASIC-style for loop. Generators are more flexible; the for loop above looks clumsy when compared with the equivalent
every write(1 to 10)

A generator operand to a non-generator forms a generator. The enclosing non-generator (such as `write()`) is re-run each time the generator suspends and resumes.

Unicon’s **every** keyword generalizes the concept of iterators found in other languages. Iterators are special control structures that walk through a collection of data. Instead of being a special feature, in Unicon iterators are just one of many ways to utilize generators. The sequence of results from an expression does not have to be stored in a structure to iterate over them. The sequence of results does not even have to be finite; many generator expressions produce an infinite sequence of results, as long as the surrounding expression keeps asking them for more. Here is another example of how **every** expressions are more flexible than for loops. The expression

```
every f(1 | 4 | 9 | 16 | 25 | 36)
```

executes the function `f` several times, passing the first few square numbers as parameters. A shorter equivalent that uses the power operator (`^`) is `every f((1 to 6)^2)`. An example in the next section shows how to generalize this to work with all the squares.

The **if**, **while**, and **every** expressions are Unicon’s primary control structures. Several other control structures are available that may be more useful in certain situations. The loop

```
until expr1 do expr2
```

is **while**’s evil twin, executing `expr2` as long as `expr1` fails; on the other hand

```
repeat expr
```

is an infinite loop, executing `expr` over and over.

There are also variations on the **if** expression introduced earlier for conditional execution. First, the **else** branch is optional, so you can have an **if** expression that does nothing if the condition is not satisfied. Second, there is a special control structure introduced for the common situation in which several alternatives might be selected using a long sequence of **if** ... **else if** ... **else if** ... **else if** ... expressions. The **case** expression replaces these chains of **if** expressions with the syntax:

```
case expr of {
  branch1: expr1
  branch2: expr2
  ...
  default: expr
}
```
When a case expression executes, the expr is evaluated, and compared to each branch value until one that matches exactly is found, as in the binary equality operator \( === \). Branch expressions can be generators, in which case every result from the branch is compared with expr. The default branch may be omitted in case expressions for which no action is to be taken if no branch is satisfied.

When we introduced the repeat loop, you probably were wondering how to exit from the loop, since most applications do not run forever. One answer would be to call one of the built-in functions that terminate the entire program when it is time to get out of the loop. The exit(), stop(), and runerr() functions all serve this valuable purpose; their differences are described in Appendix A.

A less drastic way to get out of any loop is to use the break expression:

\[
\text{break expr}
\]

This expression exits the nearest enclosing loop; expr is evaluated outside the loop and treated as the value produced by executing the loop. This value is rarely used by the surrounding expression, but the mechanism is very useful, since it allows you to chain any number of breaks together, to exit several levels of loops simultaneously. The break expression has a cousin called next that does not get out of a loop, but instead skips the rest of the current iteration of a loop and begins the next iteration of the loop.

1.10 Procedures

Procedures are a basic building block in most languages. Here is an example of an ordinary procedure. This one computes a simple polynomial, \( ax^2 + bx + c \).

\[
\text{procedure poly(x,a,b,c)}
\]
\[
\text{\quad return a \times x^2 + b \times x + c}
\]
\[
\text{end}
\]

Parameters

Procedure parameters are passed by value except for structured data types, which are passed by reference. This means that when you pass in a string, number, or cset value, the procedure gets a copy of that value; any changes the procedure makes to its copy will not be reflected in the calling procedure. On the other hand, structures that contain other values, such as lists, tables, records, and sets are not copied. The procedure being called gets a handle to the original value, and any changes it makes to the value will be visible to the calling procedure. Structure types are described in the next chapter.

When you call a procedure with too many parameters, the extras are discarded. This feature is valuable for prototyping but can be dangerous if you aren’t careful! Similarly, if
you call a procedure with too few parameters, the remaining variables are assigned the null value, &null. The null value is also passed as a parameter if you omit a parameter.

Now it is time to describe the unary operators \ and /\#. These operators test the expression against the null value. The expression /\ x succeeds and returns \ x if the value is the null value. This can be used to assign default values to procedure arguments: Any arguments that have a null value can be tested for and assigned a value. Here’s an example:

```
procedure succ(x, i)
  /i := 1
  return x + i
end
```

If this procedure is called as succ(10), the missing second parameter, i, is assigned the null value. The forward slash operator then tests i against the null value, which succeeds; therefore the value 1 is assigned to i.

The backward slash checks if its argument is non-null. For example, this will write the value of \ x if it has a non-null value:

```
write("The value of x is ", \x)
```

If \ x does in fact have the null value, then \x will fail, which will mean that the write() procedure will not be called. If it helps you to not get these two mixed up, a slash pushes its operand "up" for non-null, and "down" for a null value.

The procedure succ() shows one way to specify a default value for a parameter. The built-in functions use such default values systematically to reduce the number of parameters needed; consistent defaults for entire families of functions make them easy to remember. Another key aspect of the built-in operations is implicit type conversion. Arguments to built-in functions and operators are converted as needed.

Defaulting and type conversion are so common and so valuable in Unicon that they have their own syntax. Parameter names may optionally be followed by a coercion function (usually the name of a built in type) and/or a default value, separated by colons. These constructs are especially useful in enforcing the public interfaces of library routines that will be used by many people.

```
procedure succ(x:integer, i:integer:1)
end
```

This parameter declaration is a more concise equivalent of

```
procedure succ(x, i)
  x := integer(x) | runerr(101, x)
  /i := 1 | i := integer(i) | runerr(101, i)
end
```
Variables and scopes

Variable names used as parameters are fundamentally different from procedure names. The scope of parameters is limited to the body of a single procedure, while procedure names are visible to the entire program. Parameters and procedures are special forms of two basic kinds of variables in Unicon: local variables and global variables.

The scope of global variables, including procedure names, consists of the entire program. Their value is stored in memory for the entire execution of the program. There are two kinds of local variables; both introduce names that are defined only within a particular procedure. Regular local variables are created when a procedure is called, and destroyed when a procedure fails or returns a result; a separate copy of regular local variables is created for each call. Static local variables are global variables whose names are visible only within a particular procedure; a single location in memory is shared by all calls to the procedure, and the last value assigned in one call is remembered in the next call.

Variables do not have to be declared, and by default they are local. To get a global variable, you have to declare it outside any procedure with a declaration like this:

```unicon
global MyGlobal
```

Such a declaration can be before, after, or in between procedures within a program source file, but cannot be inside a procedure body. Of course, another way to declare a global variable is to define a procedure; this creates a global variable initialized with the appropriate procedure value containing the procedure’s instructions.

Regular and static local variables may be declared at the top of a procedure body, after the parameters and before the code starts, as in the following example:

```unicon
procedure foo()
    local x, y
    static z
    ...
end
```

Each declared local variable name may be followed by a := and an initializer expression that specifies the variable’s initial value. Without an initializer, variables start with the value &null. Although you do not have to declare local variables, large programs, library code or multi-person projects should declare all local variables. If you don’t, and some other part of the code introduces a global variable by the same name as your undeclared local, your variable will be interpreted as a reference to the global. To help avoid this problem, the -u command line option to the compiler causes undeclared local variables to produce a compilation error message.

A procedure body can begin with an initial clause, which executes only the first time the procedure is called. The initial clause is mainly used to initialize static variables in ways
that aren’t handled by initializers. For example, the following procedure returns the next number in the Fibonacci sequence each time it is called, using static variables to remember previous results in the sequence between calls.

```plaintext
procedure fib()
static x,y
local z
initial {
x := 0
y := 1
return 1
}
z := x + y
x := y
y := z
return z
end
```

**Writing your own generators**

When a procedure returns a value, the procedure and its regular local variables cease to exist. But there are expressions that don’t disappear when they produce a value: generators! You can create your own generators by writing procedures that use `suspend` instead of `return`. `suspend` is different from `return` in that it saves the point of execution within the procedure; if another value is required by the calling expression, the generator continues execution from where it previously left off.

Here is a procedure to generate all the squares. Instead of using multiplication, it uses addition to demonstrate generators! The code uses the fact that if we keep adding successive odd numbers, we get the squares.

```plaintext
procedure squares()
odds := 1
sum := 0
repeat {
suspend sum
sum +:= odds
odds +:= 2
}
end
```

To perform a computation on the squares, we can use it in an `every` statement:

```plaintext
every munge(squares())
```
Warning

This is an infinite loop! (Do you know why? Whether \texttt{munge()} succeeds or fails, \texttt{every} will always resume \texttt{squares()} for another result to try; \texttt{squares()} generates an infinite result sequence.)

The \texttt{fail} expression makes the procedure fail. Control goes to the calling procedure, returning no value, and the procedure call ceases to exist; it cannot be resumed. A procedure also fails implicitly when control flows off the end of the procedure’s body.

Here is a procedure that produces all the non-blank characters of a string, but bails out if the character \# is reached:

\begin{verbatim}
procedure nonblank(s)
  every c := !s do {
    if c == "#" then fail
    if c ~= " " then suspend c
  }
end
\end{verbatim}

Recursion

A recursive procedure is one that calls itself, directly or indirectly. There are many cases where it is the most natural way of solving the problem. Consider the famous "Towers of Hanoi" problem. Legend has it that when the universe was created, a group of monks in a temple in some remote place were presented with a problem. There are three diamond needles, and on one of them is a stack of 64 golden disks all of different sizes, placed in order with the largest one at the bottom and the smallest on top. All the disks are to be moved to a different needle under the conditions that only one disk may be moved at a time, and a larger disk can never be placed on a smaller disk. When the monks finish this task, the universe will come to an end.

How can you move the \(n\) smallest disks? If \(n\) is 1, just move it. Since it’s the smallest, this will not violate the condition. If \(n\) is greater than 1, here’s what we can do: first, move the \(n-1\) upper disks to the intermediate needle, then transfer the \(n\)th disk, then move the \(n-1\) upper disks to the destination needle. This whole procedure does not violate the requirements either (satisfy yourself that such is the case).

Now write the procedure \texttt{hanoi(n)} that computes this algorithm. The first part is simple: if you have one disk, just move it.

\begin{verbatim}
procedure hanoi(n, needle1:1, needle2:2)
  if n = 1 then write("Move disk from ", needle1, " to ", needle2)
  Otherwise, perform a recursive call with \(n-1\). First, to find the spare needle we have:

  other := 6 - needle1 - needle2
\end{verbatim}
Now move the \( n-1 \) disks from \texttt{needle1} to \texttt{other}, move the biggest disk, and then move the \( n-1 \) again. The needles are passed as additional parameters into the recursive calls. They are always two distinct values out of the set \{1, 2, 3\}.

\[
\text{hanoi}(n-1, \texttt{needle1}, \texttt{other}) \\
\text{write("Move disk from ", \texttt{needle1}, ", to ", \texttt{needle2})} \\
\text{hanoi}(n-1, \texttt{other}, \texttt{needle2})
\]

That’s it! You’re done. Listing 1-2 contains the complete program for you to try:

Listing 1-2 Towers of Hanoi

\[
\text{procedure main()} \\
\text{write("How many disks are on the towers of Hanoi?")} \\
\text{hanoi(read())} \\
\end
\]

\[
\text{procedure hanoi(n:integer, \texttt{needle1}:1, \texttt{needle2}:2)} \\
\text{local \texttt{other}} \\
\text{if n = 1 then write("Move disk from ", \texttt{needle1}, ", to ", \texttt{needle2})} \\
\text{else} \{ \\
\text{\texttt{other} := 6 - \texttt{needle1} - \texttt{needle2}} \\
\text{\texttt{hanoi}(n-1, \texttt{needle1}, \texttt{other})} \\
\text{\texttt{write("Move disk from ", \texttt{needle1}, ", to ", \texttt{needle2})}} \\
\text{\texttt{hanoi}(n-1, \texttt{other}, \texttt{needle2})} \\
\}\n\]

Turn on tracing see how this program works. To enable tracing, compile your program with a \texttt{t} option, or assign the keyword \texttt{&trace} a non-zero number giving the depth of calls to trace. Setting \texttt{&trace} to \texttt{-1} will turn on tracing to an infinite depth.

To move \( n \) disks, \( 2^n - 1 \) individual disk movements will be required. If the monks move one disk a second, it will take \( 2^{64} - 1 \) seconds, or about 60 trillion years. Wikipedia has listed the age of the universe at around 13.75 billion years. It seems unlikely that we need worry about the monks finishing their task!

**Summary**

In this chapter you have learned:

- Unicon is an expression-based language organized as a set of procedures starting from a procedure called \texttt{main()}.
- Unicon has four atomic types: arbitrary precision integers, real numbers, arbitrary length strings of characters, and character sets.
• There is no Boolean concept in Unicon; instead, control is driven by whether an expression succeeds in producing a result or fails to do so. This eliminates the need for most sentinel values, shortening many expressions.

• Generator expressions can produce multiple results as needed by the surrounding expression in order for it to produce results.

• Procedure parameters are passed by value for atomic types, and by reference for all other data types. Unicon features extensive argument defaults and automatic type coercion throughout its built-in function and operator repertoire.

• Unicon has two scope levels: global and local. Undeclared variables are implicitly defined to be local.
Chapter 2

Structures

The examples in the previous chapter employed data types whose values are *immutable*. For example, all operations that manipulate numbers and strings compute new values, rather than modify existing values. This chapter presents structured types that organize and store collections of arbitrary (and possibly mixed) types of values. When you complete this chapter, you will understand how to use these types.

- Tables associate their elements with key values for rapid lookup.
- Lists offer efficient access by position as well as by stack or queue operations.
- Records store values using a fixed number of named fields.
- Sets support operations such as union and intersection on groups of elements.
- Using structures to represent trees, graphs, and matrices.

There are several structure types that describe different basic relationships between values. The philosophy of structures in Unicon is to provide built-in operators and functions for common organization and access patterns - the flexible "super glue" that is needed by nearly all applications. Their functionality is similar to the C++ Standard Template Library or generic classes in other languages, but Unicon’s structure types are much simpler to learn and use, and are well supported by the expression evaluation mechanism described in the previous chapter.

All structure types in Icon share many aspects in common, such as the fact that structures are *mutable*. The values inside them may change. In that respect, structures are similar to a collection of variables that are bundled together. In many cases, Unicon’s structure types are almost interchangeable! Operators like subscripts and built-in functions such as `insert()` are defined consistently for many types. Code that relies on such operators is *polymorphic*: it may be used with multiple structure types in an interchangeable way.

For both the structures described in this chapter and the strings described in the next chapter, be aware that Unicon performs automatic storage management, also known as
garbage collection. If you have used a language like C or C++, you know that one of the biggest headaches in writing programs in these languages is tracking down bugs caused by memory allocation, especially dynamic heap memory allocation. Unicon transparently takes care of those issues for you.

Another big source of bugs in languages like C and C++ are pointers, values that contain raw memory addresses. Used properly, pointers are powerful and efficient. The problem is that they are easy to use incorrectly by accident; this is true for students and practicing software engineers alike. It is easy in C to point at something that is off-limits, or to trash some data through a pointer of the wrong type.

Unicon has no pointer types, but all structure values implicitly use pointer semantics. A reference is a pointer for which type information is maintained and safety is strictly enforced. All structure values are references to data that is allocated elsewhere, in a memory region known as the heap. You can think of a reference as a safe pointer: the only operations it supports are copying the pointer, or dereferencing it using an operation that is defined for its type.

Assigning a structure to a variable, or passing it as a parameter, gives that variable or parameter a copy of the reference to the structure but does not make a copy of the structure. If you want a copy of a structure, you call the function \texttt{copy(x)}, which makes a “shallow” copy of a single table, list, record, or set. If that structure contains references to other structures as its elements, those substructures are not copied by \texttt{copy()}. To copy a “deep” structure (lists of lists, tables of records, etc.) you can use the procedure \texttt{deepcopy()} that is given as an example later in this chapter.

\section{Tables}

Tables are unordered collections of values that are accessed using associated keys. They are Unicon’s most versatile type. All of the other structure types can be viewed as special cases of tables, optimized for performance on common operations. Most operations that are defined for tables are defined for other structure types as well.

Subscripts are used for the primary operations of associating keys with values that are inserted into the table, and then using keys to look up objects in the table. The \texttt{table()} function creates a new empty table. For example, the lines

\begin{verbatim}
T := table()
T["hello"] := "goodbye"
\end{verbatim}

create a new table, and associate the key "hello" with the value "goodbye". The \texttt{table()} function takes one optional argument: the default value to return when lookup fails. The default value of the default value is \&null, so after the above example, \texttt{write(T["goodbye"])} would write an empty line, since \texttt{write()} treats a null argument the same as an empty string, and \texttt{write()} always writes a newline. Assigning a value to a key that is not in the table
inserts a value into the table. This occurs in the second line of the example above, so `write(T["hello"])` writes out "goodbye".

Subscripts are the primary operation on tables, but there are several other useful operations. The `insert(T, k_1, x_1, k_2, x_2, ...)` function adds new key-value pairs to $T$. The `delete(T, k_1, k_2, ...)` function deletes values from $T$ that correspond to the supplied keys. Icon's unary * operator produces the size of its argument; for a table, $^*T$ is the number of key-value pairs in the table. Unary ! generates elements from a collection; for a table, !$T$ generates the values stored in the table. Unary ? is the random operator; for a table, ?$T$ produces a random value stored in the table. Both unary ! and ? produce values stored in a table, not the keys used to lookup values.

Function `member(T, k)` succeeds if $k$ is a key in $T$ and fails otherwise. Function `key(T)` generates the keys that have associated values. The following example prints word counts for the input (assuming `getword()` generates words of interest):

```plaintext
wordcount := table(0)
every word := getword() do wordcount[word] +:= 1
every word := key(wordcount) do write(word, " ", wordcount[word])
```

The default value for the table is 0. When a new word is inserted, the default value gets incremented and the new value (that is, 1) is stored with the new word. Tables grow automatically as new elements are inserted.

### 2.2 Lists

Lists are dynamically sized ordered collections of values. They are accessed by subscripts, with indexes starting at 1. You can also insert or remove elements from the beginning, middle, or end of the list. Lists take the place of arrays, stacks, queues, and deques found in other languages and data structures textbooks.

A list is created by calling the function `list()`, which takes optional parameters for the list's initial size and the initial value given to all elements of the list. By default `list()` creates an empty list. Alternatively, you can create a list by enclosing a comma-separated sequence of values in square brackets:

```plaintext
L := ["linux", 2.0, "unix"]
```

Lists are dynamic. Lists grow or shrink as a result of stack and queue operations. The `push()` and `pop()` functions add and remove elements from the front of a list, while `put()` and `pull()` add and remove elements at the end of the list. In addition, `insert(L, i, x)` inserts $x$ at position $i$, and `delete(L, i)` deletes the element at position $i$. The expression `[]` is another way to create an empty list; it is equivalent to calling `list()` with no arguments. The previous list could have been constructed one element at a time with the following code. `put()` accepts a variable number of arguments.
L := [ ]
put(L, "linux")
put(L, 2.0)
put(L, "unix")

Elements of the list can be obtained either through list manipulation functions or by subscripting. Given the list L above, in the following code the first line writes "unix" while the second line moves the first element to the end of the list.

write(L[3])
put(L, pop(L))

There is no restriction on the kinds of values that may be stored in a list. For example, the elements of a list can themselves be lists. You can create lists like

L := [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

and index them with multiple subscripts. L[2][3] is equivalent to L[2,3] and yields the value 6 in this example.

Lists also support several common operators. The operator *L produces the size of list L. The operators !L and ?L generate the elements of L in sequence, and produce a single random element of L, respectively. The following procedure uses the unary ! operator to sum the values in list L, which must be numbers.

procedure sum(L)
    total := 0
    every total += !L
    return total
end

Comparing the two, lists are like tables with boring keys: the positive integers starting from 1. Function member(L, k) succeeds if 0 < integer(k) <= *L, while key(L) is equivalent to the expression 1 to *L. List indexes are contiguous, unlike table keys, and so lists can support a slice operator to produce a sublist, given a pair of indexes to mark the bounds.

The L[i:j] expression produces a new list that contains a copy of the elements in L between positions i and j. The L[i+:j] expression is equivalent to L[i:i+j]. List concatenation is another valuable operator. The L1 ||| L2 expression produces a new list whose elements are a copy of the elements in L1 followed by a copy of the elements in L2.

2.3 Records

A record is a fixed-sized, ordered collection of values whose elements are accessed using user-defined named fields. A record is declared as a global name that introduces a new type with a corresponding constructor procedure, as in the following example. The field names are a comma-separated list of identifiers enclosed in parentheses.
2.4. SETS

record complex(re, im)

Record instances are created using a constructor procedure with the name of the record type. The fields of an instance are accessed by name using dot notation or string subscript, or by integer index subscript. You can use records as records, or as special tables or lists with a constant size and fixed set of keys.

member(R,s) tests whether s is a field in R; key(R) generates R’s field names. Functions like insert(), or push() are not supported on records, since they change the size of the structure that they modify. Here is a demonstration of record operations.

a := complex(0, 0)
b := complex(1, -1)
if a.re = b.re then write("not likely")
if a["re"] = a[2] then write("a.re and a.im are equal")

Unicon provides a mechanism for constructing new record types on the fly, described in Chapter 6, as well as the ability to declare classes, which are new data types that form the building blocks for object-oriented programs, described starting in Chapter 9. Records are closely related to classes and objects: they can be considered to be an optimized special case of objects that have no methods.

2.4 Sets

A set is an unordered collection of values with the uniqueness property: an element can only be present in a set once. The function set(x...) creates a set containing its arguments. For the sake of backward compatibility with Icon, list arguments to set() are not inserted; instead, the list elements are inserted. As with other structures, the elements may be of any type, and may be mixed. For example, the assignment

S := set("rock lobster", 'B', 52)

creates a set with three members: a string, a cset, and an integer. The equivalent set is produced by set(["rock lobster","B", 52]). To place a list in a set set constructor, wrap it in another list, as in set([L]), or insert the list into the set after it is created.

The functions member(), insert(), and delete() do what their names suggest. As for csets in the previous chapter, S1++S2, S1**S2, and S1--S2 are the union, intersection, and difference of sets S1 and S2. Set operators construct new sets and do not modify their operands. Because a set can contain any value, it can contain a reference to itself. This is one of several differences between Unicon sets, which are mutable structures, and mathematical sets. Another difference is that Unicon sets have a finite number of elements, while mathematical sets can be infinite in size.
As a short example, consider the following program, called `uniq`, that filters duplicate lines in its standard input as it writes to its standard output. Unlike the UNIX utility of this name, our version does not require the duplicate lines to be adjacent.

```plaintext
procedure main()
    S := set()
    while line := read() do
        if not member(S, line) then {
            insert(S, line)
            write(line)
        }
    end
```

Sets are closely related to the table data type. They are very similar to an optimized special case of tables that map all keys to the value `&null`. Unlike tables, sets have no default value and do not support the subscript operator.

### 2.5 Using Structures

Structures can hold other structures, allowing you to organize information in whatever way best fits your application. Building complex structures such as a table of lists, or a list of records that contain sets, requires no special trickery or new syntax. Examples of how such structures are accessed and traversed will get you started. Recursion is often involved in operations on complex structures, so it plays a prominent role in the examples. The concept of recursion was discussed in Chapter 1.

### A Deep Copy

The built-in function `copy(x)` makes a one-level copy of structure values. For a multi-level structure, you need to call `copy()` for each substructure if the new structure must not point into the old structure. This is a natural task for a recursive function.

```plaintext
procedure deepcopy(x)
    local y
    case type(x) of {
        "table"|"list"|"record": {
            y := copy(x)
            every k := key(x) do y[k] := deepcopy(x[k])
        }
        "set": {
            y := set()
            every insert(y, deepcopy(!x))
        }
    }
```
This version of `deepcopy()` works for arbitrarily deep tree structures, but the program execution will crash if `deepcopy()` is called on a structure containing cycles. It also does not copy directed acyclic graphs correctly. In both cases the problem is one of not noticing when you have already copied a structure, and copying it again. The Icon Program Library has a deep copy procedure that handles this problem, and we present the general technique that is used to solve it in the next section.

### Representing Trees and Graphs

Since there is no restriction on the types of values in a list, they can be other lists too. Here is an example of how a tree may be implemented with records and lists:

```plaintext
record node(name, links)
...
barney := node("Barney", list())
betty := node("Betty", list())
bambam := node("Bam-Bam", [barney, betty])
```

The structure created by these expressions is depicted in Figure 2-1. The list of links at each node allows trees with an arbitrary number of children at the cost of extra memory and indirection in the tree traversals. The same representation works for arbitrary graphs.

![Figure 2-1: A Record Containing a List of Two Records](image)

To find every node related to variable `bambam`, follow all the links reachable starting from `bambam`. Here is a procedure that performs this task.
procedure print_relatives(n)
local i
static relatives
initial relatives := set()
every i := n | ln.links do {
    if not member(relatives, i.name) then {
        write(i.name)
        insert(relatives, i.name)
        print_relatives(i)
    }
}
}

Calling print_relatives(bambam) will print

Bam-Bam
Barney
Betty

Static variables and the initial clause are explained in Chapter 1. Can you guess what purpose static variable relatives serves? For a proper tree structure, it is not needed at all, but for more general data structures such as directed graphs this static variable is very important! One defect of this procedure is that there is no way to reset the static variable and call print_relatives() starting from scratch. How would you remove this defect?

The n-Queens Example

The 8-Queens problem is a classic backtracking problem. The goal is to place eight queens on a chessboard so that none of the queens attack any other. Here is a solution to a more general form of the problem, that of placing n queens on an n x n board. The solution we present is by Steve Wampler, and it is in the Icon Program Library.

An array of size n stores the solutions, with each element representing a column. The values in the array are integers specifying the row in each column that has the queen. (Since the queens cannot attack each other, each column must contain exactly one queen.) The problem size n and the array are declared global so that all procedures can see them; this allows the program to avoid passing these variables in to every procedure call. Use globals sparingly, and only where they are appropriate, as is the case here.

link options
global solution, n
procedure main(args)
local i, opts
2.5. USING STRUCTURES

The program starts by handling command-line arguments. In Unicon programs, `main()` is called with a single parameter that is a list of strings whose elements are the command-line arguments of the program.

The n-queens program recognizes only one thing on the command line: the option `-n` followed by an integer specifies the size of board to use. Thus the command line `queens -n 9` will generate solutions on a 9x9 board. The default value of `n` is 6. The `options()` procedure is an Icon Program Library procedure described in Appendix B; it removes options from the command line and places them in a table whose keys are option letters such as "n". Library procedures such as `options()` are incorporated into a program using the `link` declaration, as in the `link options` that begins the code fragment above. A link declaration adds the procedures, global variables, and record types in the named module (in this case, procedure `options()` came from a file `options.icn`) to the program.

```plaintext
opts := options(args,"n+")
n := \opts["n"] | 6
if n <= 0 then stop("-n needs a positive numeric parameter")
```

The value `n` gives the size for the solution array and also appears in a banner:

```plaintext
solution := list(n)  # a list of column solutions
write(n,"-Queens:")
every q(1)  # start by placing queen in first column
end
```

Now comes the meat of the program, the procedure `q(c)`. It tries to place a queen in column `c` and then calls itself recursively to place queens in the column to the right. The `q(c)` procedure uses three arrays: `rows`, `up`, and `down`. They are declared to be `static`, meaning that their values will be preserved between executions of the procedure, and all instances of the procedure will share the same lists. Since each row must have exactly one queen, the rows array helps to make sure any queen that is placed is not on a row that already has a queen. The other two arrays handle the diagonals: `up` is an array (of size `2n-1`) of the upward slanting diagonals, and `down` is an array for the downward slanting diagonals. Two queens in positions `(r_1, c_1)` and `(r_2, c_2)` are on the same "up" diagonal if `n+r_1-c_1 = n+r_2-c_2` and they are on the same "down" diagonal if `r_1+c_1-1 = r_2+c_2-1`. Figure 2-2 shows some of the "up" and "down" diagonals.
Figure 2-2: Up and Down Diagonals in the n-Queens Problem

```
# q(c) - place a queen in column c.
#
procedure q(c)
local r
static up, down, rows
initial {
  up := list(2*n-1,0)
  down := list(2*n-1,0)
  rows := list(n,0)
}
```

The next expression in \texttt{q()} is an \texttt{every} loop that tries all possible values for the queen in row \texttt{c}. The variable \texttt{r} steps through rows 1 to 8. For any row at which the program places a queen, it must ensure that

1. \texttt{rows[r]} is zero, that is, no other column has a queen in row \texttt{r},
2. \texttt{up[n+r-c]} is 0, that is, there is not already a queen in the ”up” diagonal, and
3. \texttt{down[r+c-1]} is 0, that is, there is not already a queen in the down diagonal.

If these conditions are met, then it is OK to place a queen by assigning a 1 to all those arrays in the appropriate position:

\[
\text{every } 0 = \texttt{rows[r := 1 to n]} = \texttt{up[n+r-c]} = \texttt{down[r+c-1]} \&\ \\
\texttt{rows[r] <- up[n+r-c] <- down[r+c-1] <- 1 do } \texttt{ }
\]
For assignment, instead of := this expression uses the *reversible assignment* operator <-. This assigns a value just like in conventional assignment, but it remembers the old value; if it is ever resumed, it restores the old value and fails. This causes the appropriate entries in the row, up, and down arrays will be reinitialized between iterations.

When the every loop found a good placement for this column, either the program is done (if this was the last column) or else it is time to try to place a queen in the next row:

```unicon
solution[c] := r       # record placement.
if c = n then show()
else q(c + 1)        # try to place next queen.
}
end
```

That’s it! The rest of the program just prints out any solutions that were found.

Printing the chess board is similar to other reports you might write that need to create horizontal lines for tables. The repl() function is handy for such situations. The repl(s, i) function returns i ”replicas” of string s concatenated together. The show() function uses it to create the chessboard.

```unicon
# # show the solution on a chess board.
# procedure show()
static count, line, border
initial {
    count := 0
    line := repl("| ",n) || "|
    border := repl("----",n) || "-"
}
write("solution: ", count+:=1, "\n ", border)
every line[4*(isolution - 1) + 3] <- "Q" do {
    write(" ", line, "\n ", border)
}
write()
end
```

### 2.6 Summary

Unicon’s structures are better than sliced bread. To be fair, this is because Icon’s inventors really got things right. These structures are the foundations of complex algorithms and the glue that builds sophisticated data models. They are every computer scientists’ buzzword-compliant best friends: polymorphic, heterogeneous, implicitly referenced, cycle-capable,
dynamically represented, and automatically reclaimed. They provide a direct implementation of the common information associations used in object-oriented design. But most important of all, they are extremely simple to learn and use.
Chapter 3
String Processing

In addition to its groundbreaking expression evaluation, Unicon inherits other compelling features from Icon that reduce the effort required to write complex programs. Icon’s ancestor is SNOBOL4, the grandfather of all string processing languages, and from it come some of the most flexible and readable built-in string processing facilities found in any language.

In this chapter you will learn

- How to manipulate strings and sets of characters
- Icon’s string scanning control structure
- How to write custom pattern matching primitives, with backtracking

Techniques for matching regular expressions and context free grammars

3.1 String Indexes

You have already seen string literals delimited by double quotes, and the most common operators that work on strings: the size of a string is given by the unary * operator, substrings can be picked out with indexes, and two strings can be concatenated with the || operator. Now it is time to present a deeper understanding of the meaning of indexes as they are used with strings and lists.

Indexes in a string refer to the positions between characters. The positions are numbered starting from 1. The index 0 refers to the position after the last character in the string, and negative indices count from the right side of the string:
The expression $s[i:j]$ refers to the substring of $s$ that lies between positions $i$ and $j$. If either $i$ or $j$ is not a valid index into $s$, the expression fails. The expression $s[k]$ is short for $s[k:k+1]$ and refers to a single character at position $k$. The expression $s[k:+n]$ is the substring of length $n$ starting at position $k$. If $s$ is the string "hello, world!" then the expressions

$$s[7] := "\text{puny}\"$$
$$s[13:18] := "\text{earthlings}\"$$

c change $s$ into "hello, puny earthlings!", illustrating the ease with which insertions and substitutions are made. The first assignment changes the string to "hello, puny world!", replacing a single character with six characters and thereby increasing its length. The second assignment operates on the modified string, replacing "world" with "earthlings".

Strings are values, just like numbers; if you copy a string and then work on the copy, the original will be left unchanged:

```plaintext
s := "string1"
new_s := s
new_s[7] := "2"
```

Now the value of new_s is "string2" but s is left unchanged.

As mentioned in Chapter 1, strings can be compared with string comparison operators such as $==$.

```plaintext
if line[1] == "#" then ...
```

If you find you are writing many such tests, the string processing you are doing may be more cleanly handled using the string scanning facilities, described below. But first, here is some more detail on the character set data type, which is used in many of the string scanning functions.

### 3.2 Character Sets

A cset is a set of characters. It has the usual properties of sets: order is not significant, and a character can only occur once in a cset. A cset literal is represented with single quotes:

```plaintext
c := 'aeiou'
```

Since characters can only occur once in a cset, duplicates in a cset literal are ignored; for example, 'aaiiee' is equivalent to 'aie'. Strings can be converted to csets and vice versa. Since csets do not contain duplicates, when a string is converted to a cset, all the duplicates are removed.

Therefore to see if a string is composed of all the vowels and no consonants:
if cset(s) == 'aeiou' then ...

Or, to find the number of distinct characters in a string:

\n := *cset(s)

The ! operator generates the members of a cset in sorted order; this is also useful in some situations.

### 3.3 Character Escapes

Both strings and csets rely on the backslash as an escape character within string literals. A backslash followed by an *escape code* of one or more characters specifies a non-printable or control character. Escape codes may be specified by a numeric value given in hex or octal format - for example, "\x41". Alternatively, any control character may be specified with an escape code consisting of the caret (\^) followed by the alphabetic letter of the control character. A cset containing control-C, control-D, and control-Z could be specified as "\^c\^d\^z". For the most common character escapes, a single-letter code is defined, such as "\t" for the tab character, or "\n" for the newline. For all other characters, the character following the backslash is the character; this is how quotes or backslashes are included in literals. The escape codes are summarized in Table 3-1.

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\d</td>
<td>delete</td>
</tr>
<tr>
<td>\e</td>
<td>escape</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>\l</td>
<td>line feed</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
</tr>
<tr>
<td>'</td>
<td>quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
</tr>
<tr>
<td>\</td>
<td>backslash</td>
</tr>
<tr>
<td>\000</td>
<td>octal</td>
</tr>
<tr>
<td>\xhh</td>
<td>hexadecimal</td>
</tr>
<tr>
<td>\x</td>
<td>Control-x</td>
</tr>
</tbody>
</table>

### 3.4 String Scanning

Icon’s string analysis facility is called *string scanning*. A scanning environment consists of a string *subject* and an integer *position* within the subject at which scanning is to be performed. These values are held by the keyword variables \&subject and \&pos. Scanning environments are created by an expression of the form

\n s ? expr
The binary ? operator sets the subject to its left argument and initializes the position to 1; then it executes the expression on the right side.

The expression usually has matching functions in it. Matching functions change the position, and return the substring between the old and new positions. For example: move(j) moves the position j places to the right and returns the substring between the old and new position. This string will have exactly j characters in it. When the position cannot move as directed, for example because there are less than j characters to the right, move() fails. Here is a simple example:

    text ? {  
        while move(1) do  
            write(move(1))  
    }

This code writes out every other character of the string in variable text.

Another function is tab(i), which sets the position &pos to its argument and returns the substring that it passed over. So the expression tab(0) will return the substring from the current position to the end of the string, and set the position to the end of the string.

String scanning functions examine a string and generate the interesting positions in it. We have already seen find(), which looks for substrings. In addition to the other parameters that define what the function looks for, these string functions end with three optional parameters: a string to examine and two integers. These functions default their string parameter to &subject, the string being scanned. The two integer positions specify where in the string the processing will be performed; they default to 1 and 0 (the entire string), or &pos and 0 if the string defaulted to use &subject. Here is a generator that produces the words from the input:

    procedure getword()  
        local wchar, line  
        wchar := &letters ++ &digits ++ '\-'  
        while line := read() do  
            line ? while tab(upto(wchar)) do {  
                word := tab(many(wchar))  
                suspend word  
            }  
        end

Variable wchar is a cset of characters that are allowed in words, including apostrophe (which is escaped) and hyphen characters. upto(c) returns the next position at which a character from the cset c occurs. The many(c) function returns the position after a sequence of characters from c, if one or more of them occur at the current position. The expression tab(upto(wchar)) advances the position to a character from wchar; then tab(many(wchar))
moves the position to the end of the word and returns the word that is found. This is a
generator, so when it is resumed, it takes up execution from where it left off and continues
to look for words (reading the input as necessary).

Notice the first line: the cset wchar is the set union of the upper- and lowercase letters
(the value of the keyword &letters) and the digits (the keyword &digits). This cset union is
performed each time getword() is called, which is inefficient. Instead, the procedure ought
to calculate the value and store it for all future calls to getword().

To do this, you declare the variable to be static, causing its value to persist across
calls to the procedure. Normal local variables are initialized to the null value each time a
procedure is entered. To do this, add these two lines to the beginning of the procedure:

    static wchar
    initial wchar := &letters ++ &digits ++ '\-'

The match(s) function takes a string argument and succeeds if s is found at the current
position in the subject. If it succeeds, it produces the position at the end of the matched
substring. This expression

    if tab(match("-")) then sign := -1 else sign := 1

looks to see if there is a minus sign at the current position; if one is found, &pos is moved
past it and the variable sign is assigned a -1; otherwise, it gets a 1. The expression

    tab(match(s))

occurs quite often in string scanning, so it is given a shortcut: =s does the
same thing.

The last two string scanning functions to round out Icon’s built-in repertoire are any(c)
and bal(c1,c2,c3). any(c) is similar to many(), but only tests a single character being
scanned to see if it is in cset c. The bal() function produces positions at which a character
in c1 occurs, similar to upto(), with the added stipulation that the string up to those
positions is balanced with respect to characters in c2 and c3. A string is balanced if it
has the same number of characters from c2 as from c3 and there are at no point more c3
characters present than c2 characters. The c1 argument defaults to &cset. Since c2 and
c3 default to '(' and ')', bal() defaults to find balanced parentheses.

The restriction that bal() only returns positions at which a character in c1 occurs is a
bit strange. Consider what you would need to do in order to write an expression that tells
whether a string s is balanced or not.

You might want to write it as s ? (bal() = *s+1) but bal() will never return that position.
Concatenating an extra character solves this problem:

    procedure isbalanced(s)
      return (s || "") ? (bal() = *s+1)
    end
If string $s$ is very large, this solution is not cheap, since it creates a new copy of string $s$. You might write a version of `isbalanced()` that doesn’t use the `bal()` function, and see if you can make it run faster than this version. An example later in this chapter shows how to use `bal()` in a more elegant manner.

### File Completion

Consider the following gem, attributed to Jerry Nowlin and Bob Alexander. Suppose you want to obtain the full name of a file, given only the first few letters of a filename and a list of complete filenames. The following one line procedure does the trick:

```unicon
procedure complete(prefix, filenames)
    suspend match(prefix, p := !filenames) & p
end
```

This procedure works fine for lists with just a few members and also for cases where `prefix` is fairly large.

### Backtracking

The matching functions we have seen so far, (`tab()` and `move()`), are actually generators. That is, even though they only produce one value, they suspend instead of returning. If expression evaluation ever resumes one of these functions, they restore the old value of `&pos`. This makes it easy to try alternative matches starting from the same position in the string:

```unicon
s ? (="0x" & tab(many(&digits ++ 'abcdefABCDEF'))) |
    tab(many(&digits))
```

This expression will match either a hexadecimal string in the format used by C or a decimal integer. Suppose $s$ contains the string "0xy". The first part of the expression succeeds and matches the "0x"; but then the expression `tab(many(&digits ++ 'abcdef'))` fails; this causes Unicon to resume the first `tab()`, which resets the position to the beginning of the string and fails. Unicon then evaluates the expression `tab(many(&digits))` which succeeds (matching the string "0"); therefore the entire expression succeeds and leaves `&pos` at 2.

**Warning**

Be careful when using `tab()` or `move()` in a surrounding expression that can fail! The fact that `tab()` and `move()` reset `&pos` upon expression failure causes confusion and bugs when it happens accidentally.
Concordance Example

Listing 3-1 illustrates the above concepts and introduces a few more. Here is a program to read a file, and generate a concordance that prints each word followed by a list of the lines on which it occurs. Short words like "the" aren’t interesting, so the program only counts words longer than three characters.

Listing 3-1 A simple concordance program

```plaintext
procedure main(args)
    (*args = 1) | stop("Need a file!")
    f := open(args[1]) | stop("Couldn’t open ", args[1])
    wordlist := table()
    lineno := 0

    while line := map(read(f)) do {
        lineno +:= 1
        every word := getword(line) do
            if *word > 3 then {
                # if word isn’t in the table, set entry to empty list
                /wordlist[word] := list()
                put(wordlist[word], lineno)
            }
        }
    L := sort(wordlist)
    every l := !L do {
        writes(l[1], 	"
        linelist := ""
        # Collect line numbers into a string
        every linelist ||:= (!l[2] || ", ")
        # trim the final ", 
        write(linelist[1:-2])
    }
end

procedure getword(s)
    s ? while tab(upto(&letters)) do {
        word := tab(many(&letters))
        suspend word
    }
end
```

If we run this program on this input:

```
Half a league, half a league,
```
Half a league onward,
All in the valley of Death
Rode the six hundred.

the program writes this output:

death 3
half 1, 2
hundred 4
league 1, 1, 2
onward 2
rode 4
valley 3

First, note that the main() procedure requires a command-line argument, the name of a file to open. Also, we pass all the lines read through the function map(). This is a function that takes three arguments, the first being the string to map; and the second and third specifying how the string should be mapped on a character by character basis. The defaults for the second and third arguments are the uppercase letters and the lowercase letters, respectively; therefore, the call to map() converts the line just read in to all lowercase.

3.5 Regular Expressions

The Icon Program Library (included with the Unicon distribution) provides regular expression matching functions. To use it, add the line link regexp at the top of the program. Listing 3-2 is an example of a search-and-replace program called (somewhat inappropriately) igrep.icn: The actual searching and replacing is performed on each line of text by procedure re_sub(). This procedure illustrates many classic aspects of string scanning. It marches through the string from right to left using a while loop. It builds up a result string, which by default would be a copy of its scanned string. At the start of each occurrence of the regular expression, the replacement string is appended to the result, and the regular expression is tabbed over and not appended to the result. When no more occurrences of the regular expression are found, the remainder of the string is appended to the result.

Listing 3-2 A simple grep-like program

link regexp
procedure main(av)
local f, re, repl
every (f|re|repl) := pop(av)
f := open(f) | stop("can't open file named: ", f)
while line := read(f) do
write(re_sub(line, re, repl))
end
procedure re_sub(str, re, repl)
  result := ""
  str ? {
    while j := ReFind(re) do {
      result ||:= tab(j) || repl
      tab(ReMatch(re))
    }
    result ||:= tab(0)
  }
  return result
end

To replace all occurrences of "read\|write" with "IO operation" you could type

`igrep mypaper.txt "read\|write" "IO Operation"`

Since the program has access to the pattern matching operation at a finer grain, more complex operations are possible, this search-and-replace is just an example.

3.6 Grammars

Grammars are collections of rules that describe syntax, the combinations of words allowed in a language. Grammars are used heavily both in linguistics and in computer science. Pattern matching using a grammar is often called parsing, and is one way to match patterns more complex than regular expressions can handle. This section presents some simple programming techniques for parsing context free grammars. Context free grammars utilize a stack to recognize a fundamentally more complex category of patterns than regular expressions can; they are defined below.

For linguists, this treatment is elementary, but introduces useful programming techniques. For writers of programming language compilers, an automatic parser generator tool that you can use with Unicon or Icon is described in Chapter 18. If you are not interested in grammars, you can skip the rest of this chapter.

A context-free grammar or CFG is a set of rules or productions. Here is an example:

\[
S \rightarrow S S \\
\mid (S) \\
\mid ()
\]

This grammar has three productions. There are two kinds of symbols, non-terminals like \( S \) that can be replaced by the string on the right side of a rule, and terminals like \( ( \) and \( ) \). An application of a production rule is called a derivation. One special non-terminal
CHAPTER 3. STRING PROCESSING

is called the *start symbol*; a string is accepted by the grammar if there is a sequence of derivations from the start symbol that leads to the string. By convention the start symbol is the first non-terminal in the definition of the grammar. (This grammar only has one non-terminal, and it is also the start symbol.)

This grammar matches all strings of balanced parentheses. The string \((())()())\) can be matched by this derivation:

\[
\begin{align*}
S \rightarrow & (S) \rightarrow (SS) \rightarrow (()S) \rightarrow ((S)) \rightarrow \\
& ((SS)) \rightarrow (()(S)) \rightarrow (()())
\end{align*}
\]

**Parsing**

Unicon can parse grammars in a natural way using matching functions. A production

\[
A \rightarrow B \ a \ D \\
| \ C \ E \ b
\]

can be mapped to this matching function:

```unicon
procedure A()
    suspend (B() & =$a$ & D()) | (C() & E() & =$b$)
end
```

This procedure first tries to match a string matched by $B$, followed the character $a$, followed by a string matched by $D$. If $D$ fails, execution backtracks across the =$a$ (resetting &pos) and resume $B()$, which will attempt the next match.

If the sub-expression to the left of the alternation fails, then execution will try the sub-expression on the right, $C() & E() & =$b$ until something matches - in which case $A$ succeeds, or nothing matches - which will cause it to fail.

Parsers for any CFG can be written in this way. However, this is an expensive way to do it! Unicon’s expression evaluation will try all possible derivations trying to match a string. This is not a good way to parse, especially if the grammar is amenable to lookahead methods. A more efficient method is given in the next section. For serious parsing jobs, Chapter 18 shows how to use the Unicon versions of the standard industrial-strength lexical analyzer and parser generation tools, lex and yacc.

**Doing It Better**

Many grammars can be parsed more efficiently using well-known techniques - consult a book on compilers for details. Here is one way of parsing a grammar using some of the built-in functions. Consider this grammar for an arithmetic expression:
E -> T | T + E  
T -> F | F * T  
F -> a | b | c | ( E )

Listing 3-3 is an Unicon program that recognizes strings produced by this grammar:

Listing 3-3 Expression parser

procedure main()
    while line := read() do
        if expr(line) == line then write("Success!")
        else write("Failure.")
    end
procedure expr(s)
    s ? {
        while t := tab(bal('+')) do {
            term(t) | fail ; ="+"
        }
        term(tab(0)) | fail
    }
    return s
end
procedure term(s)
    s ? {
        while f := tab(bal('*')) do {
            factor(f) | fail ; ="*"
        }
        factor(tab(0)) | fail
    }
    return s
end
procedure factor(s)
    s ? suspend ="a" | ="b" | ="c" | ( ="(" || expr(bal(''))) || =")")
end

The interesting procedure here is bal(). With ')' as its first argument, bal() scans to the closing parenthesis, skipping over any parentheses in nested subexpressions, which is exactly what is needed here.

The procedure factor() is written according to the rule in the previous section. The procedures expr() and term() have the same structure. The expr() procedure skips any subexpressions (with balanced parentheses) and looks for a +. We know that this substring is a well-formed expression that is not a sum of terms, therefore, it must be a term. Similarly term() looks for * and it knows that the expression does not contain any * operators at the same nesting level; therefore it must be a factor.
Notice that the procedures return the strings that they matched. This allows us to check if the whole line matched the grammar rather than just an initial substring. Also, notice that \texttt{factor()} uses string concatenation instead of conjunction, so that it can return the matched substring.

\section*{Summary}

Unicon’s string processing facilities are extensive. Simple operations are very easy, while more complex string analysis has the support of a special control structure, string scanning. String scanning is not as concise as regular expression pattern matching, but it is fundamentally more general because the code and patterns are freely intermixed.
Chapter 4

Advanced Language Features

The previous chapters described a wide range of built-in computational facilities that help to make Unicon a great language. This chapter delves into interesting features that make Unicon more than just the sum of its parts. This chapter demonstrates:

- Controlling expressions more precisely
- Using list structures and procedure parameter lists interchangeably
- Holding a generator expression in a value so that its results can be used in different locations throughout the program
- Defining your own control structures
- Evaluating several generator expressions in parallel
- Permuting strings using sophisticated mappings

4.1 Limiting or Negating an Expression

Chapter 1 described generators and the expression mechanism without mentioning many methods for using them, other than every loops. Suppose you wish to generate five elements from a table. If the table has thousands of elements, then you may want to generate just five elements precisely in a situation where generating all the table elements with !T is infeasible. You could write an every loop that breaks out after five iterations, but this solution isn’t easy to use within some more complex expressions. The binary backslash operator expr \ i limits expr to at most i results. If expr has fewer results, the limitation operator has no effect; once i results have been obtained, limitation causes the expression to fail even if it could produce more results.

Unicon does not have a boolean type, so Chapter 1 downplayed the standard logical operators. The alternation operator (|) resembles a short-circuit OR operator, since it generates its left operand and only evaluates its right operand if the left operand or the surrounding expression fails. The conjunction operator (&) resembles a short-circuit AND
CHAPTER 4. ADVANCED LANGUAGE FEATURES

operator; it evaluates its left operand, and if that operand succeeds, the result of the
conjunction is the result of its right operand. The reserved word not rounds out the boolean-
like operators. If expr produces no results, then not expr will succeed (and produce a null
value); if expr produces any results, then the not expression fails. The not operator can
remedy certain forms of generator confusion. Compare the following two expressions:

\[
\begin{align*}
\text{if not (s == ("good"|"will"|"hunting")) then write("nope")}
\text{if (s \sim== ("good"|"will"|"hunting")) then write("uh huh")}
\end{align*}
\]

The first expression uses not to ensure that string s is none of the three words. The
second expression always writes "uh huh", because any string s that you pick will be not
equal (\sim==) to at least one of the three strings in the alternation. The then part will always
execute, which is probably not what was intended.

Note

Negating an == operator is not the same as using a \sim== operator!

The conjunction operator expr₁ & expr₂ has an alternate syntax, a comma-separated list
of expressions in parentheses: (expr₁, expr₂). Any number of expressions may be present;
the whole expression succeeds if they all succeed. This looks similar to procedure call
syntax because it is similar: a procedure call mutually evaluates all the actual parameters
before the procedure is invoked. Invocation allows a string or integer in the “procedure”
slot in front of a parenthesized argument list. For a string, as in s(x), a procedure by the
name given in s is called; if s had the value "foo", then s(x) is the same as foo(x). For an
integer value i, after all arguments are evaluated, the value of the entire expression is the
value of the i’th argument.

4.2 List Structures and Parameter Lists

The functions write() and put() take any number of arguments; this flexibility is powerful
and convenient. You can write variable argument procedures of your own by ending the
last parameter in your procedure declaration with empty square brackets:

\[
\text{procedure myfunc(x, y, z[])}
\]

In this case, instead of throwing away all arguments after the third, the third parameter
and all parameters that follow are placed into a newly-constructed list. A call to the above
procedure with myfunc(1, 2, 3, 4, 5) causes z to have the value [3, 4, 5].

It is also useful to do the opposite and construct a list of dynamic (or user-supplied)
length, and then call a procedure with that list as its parameter list. The apply operator,
binary ! performs this feat. If you call write! L, then all the elements of L are written
contiguously on a single line (unless they contain newline characters).
4.3 Co-expressions

A co-expression is an independent, encapsulated thread-like context, where the results of an expression (hopefully a generator!) can be picked off one at a time. Suppose you are writing a program that generates code, and you need something that will generate unique variable names. This expression will generate names:

"name" || seq()

Function seq() produces an infinite sequence of integers, by default starting at 1, so the whole expression generates "name1", "name2", "name3", ... You can use this expression anywhere in your code; but you may need it in several different places.

There are times when you need to separate the evaluation of an expression from its location in the program. The normal way to do this would be a procedure. You can make separate calls to a procedure from different locations in your program, but there is no easy way to use the results from a single instance of a generator in multiple locations. You can put all the results in a list (not a good idea for generators with infinite result sequences) or rewrite the procedure to produce the sequence using separate calls, but this requires static or global variables, and is awkward at best:

```
procedure nameseq()
    static i
    initial i := 0
    return "name" || (i +=: 1)
end
```

Now, consider the code generating program again. It may need not one name sequence, but two kinds of names: statement labels and temporary variables. It would be poor engineering to write a different procedure for each such sequence. The nameseq() procedure was already cumbersome for so simple a task, but generalizing it for multiple kinds of names makes it really messy. By creating a pair of co-expressions, you can capture exactly what is needed with a lot less code:

```
labelname := create("_L" || seq())
varname := create("_V" || seq())
```

In both cases, create expr allocates and initializes an evaluation context plus the memory needed to evaluate expression expr, but does not start to evaluate it. Since the co-expression value may be used outside the procedure call where it is created, the evaluation context includes a copy of the local variables and parameters used in the expression. When a co-expression is activated, it produces the next value. A co-expression is activated by the @ operator. Each activation of labelname produces the next string in the sequence "_L0", "_L1", "_L2", and so on. Similarly, each activation @varname produces the next in the sequence "_V0", "_V1", "_V2", and so on.
loop_name := @labelname
tempvar_name := @varname

After a co-expression has produced all its results, further evaluation with @ will fail. The ^ operator produces a new co-expression with the same expression as its argument, but "rewound" to the beginning.

\[ c := ^c \]

### 4.4 User-Defined Control Structures

Control structures are language elements that determine in what order, and how many times, expressions are executed. Co-expressions are used to implement new control structures when procedures that take co-expression parameters control the order and number of times they are activated. Consider a control structure that selects values from the first expression at positions specified by the second. This could be called as:

\[ \text{seqsel([create fibonacci(), create primes()])} \]

Assuming that you have a pair of generator procedures that produce the Fibonacci numbers \(1, 1, 2, 3, 5, 8, 13, \ldots\) and the primes \(2, 3, 5, 7, 11, \ldots\), this expression produces the numbers 1, 2, 5, 13, 89, .... Here is the implementation of seqsel():

```unicon
procedure seqsel(a)
   (*a = 2) | stop("seqsel requires a list of two arguments")
   e1 := a[1]; e2 := a[2]
   # position in the first stream we are looking at
   index := 1
   repeat {  
      # Get the next index
      (i := @e2) | fail
      # Keep getting values from the second expression until
      # we get to the i'th one. If e1 cannot produce that
      # many values, we fail.
      every index to i do
         (value := @e1) | fail
         suspend value
         index := i+1
      }
end
```

Unicon provides a syntactic short-cut for this kind of usage:

```unicon
proc([create e1, create e2, ..., create en])
```

can also be written with curly brackets, as

```unicon
proc{e1, e2, ..., en}
```
4.5 Parallel Evaluation

Co-expressions can be used to evaluate expressions "in parallel". This program writes a table of ASCII characters with the hex, decimal, and octal equivalents:

```plaintext
procedure main()
    dec := create(0 to 255)
    hex_dig := "0123456789abcdef"
    hex := create(!hex_dig || !hex_dig)
    oct := create((0 to 3) || (0 to 7) || (0 to 7))
    char := create image(!&cset)
    while write(@dec, \t, @oct, \t, @hex, \t, @char)
end
```

Co-expression `dec` produces the sequence 0, 1, 2, ... 255; `hex` the sequence "00", "01", "03", ..., "ff"; `oct` the sequence "001", "002", ..., "377"; and `char` the sequence ..., " ", ",!", ..., "A", ..., "Z", ..., "a", ..., "z", and so forth.

Every invocation of `write()` results in all the co-expressions being activated once, so they are all run in lock-step, producing this table:

<table>
<thead>
<tr>
<th>0</th>
<th>000</th>
<th>00</th>
<th>&quot;\x00&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
<td>01</td>
<td>&quot;\x01&quot;</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>02</td>
<td>&quot;\x02&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>45</td>
<td>055</td>
<td>2d</td>
<td>&quot;.-&quot;</td>
</tr>
<tr>
<td>46</td>
<td>056</td>
<td>2e</td>
<td>&quot;..&quot;</td>
</tr>
<tr>
<td>47</td>
<td>057</td>
<td>2f</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>48</td>
<td>060</td>
<td>30</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>49</td>
<td>061</td>
<td>31</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>50</td>
<td>062</td>
<td>32</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>90</td>
<td>132</td>
<td>5a</td>
<td>&quot;Z&quot;</td>
</tr>
<tr>
<td>91</td>
<td>133</td>
<td>5b</td>
<td>&quot;[&quot;</td>
</tr>
<tr>
<td>92</td>
<td>134</td>
<td>5c</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>93</td>
<td>135</td>
<td>5d</td>
<td>&quot;]&quot;</td>
</tr>
<tr>
<td>94</td>
<td>136</td>
<td>5e</td>
<td>&quot;^&quot;</td>
</tr>
<tr>
<td>95</td>
<td>137</td>
<td>5f</td>
<td>&quot;_&quot;</td>
</tr>
<tr>
<td>96</td>
<td>140</td>
<td>60</td>
<td>&quot;11&quot;</td>
</tr>
<tr>
<td>97</td>
<td>141</td>
<td>61</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>255</td>
<td>377</td>
<td>ff</td>
<td>&quot;\xff&quot;</td>
</tr>
</tbody>
</table>
```

Parallel evaluation can also be used to assign to a set of variables:
ce := create !stat(f) every (dev | ino | mode | lnk | uid | gid) := @ ce

Note
stat() returns file information. It is presented in the next chapter.
Co-expressions can be expensive. This is probably not a good way to assign a series of values to a group of variables but it demonstrates an interesting technique.

4.6 Coroutines

In conventional invocation, procedures have an asymmetric relationship; when control is transferred from the caller to the callee, the callee procedure starts execution at the top. Coroutines have an equal relationship: when control is transferred from one coroutine to another, execution starts from the point that execution was suspended. This process is called resumption. The producer/consumer problem is a good example of procedures that have an equal relationship. Figure 4-1 shows how the control flow between coroutines is different from that of conventional procedures.

![Figure 4-1: The Difference Between Procedures and Coroutines](image)

Can you tell what the next example computes from its integer command-line argument?

Listing 4-1
Producer and Consumer Coroutines

procedure main(args)
C1 := create consumer(args[1])
4.7 PERMUTATIONS

C2 := create producer(C1)
   @C2
end

procedure producer(ce)
   x := 1
   repeat {
      val := x ^ 2
      ret := val @ ce | break
      x +:= 1
   }
   @ &main
end

procedure consumer(limit)
   value := @ &source
   repeat {
      # process value
      if value > limit then break
      if value % 2 = 0 then write(value)
      value := retval @ &source
   }
end

When producer resumes consumer, it passes value; the consumer passes a return code (retval) back. &source is the coexpression that activated the current co-expression.

Note

This example doesn’t mean the producer/consumer problem should always be done with coroutines!

4.7 Permutations

We have seen one usage of map(), where it transformed mixed-case strings to all lowercase. In that type of usage, the first string is the one that we are manipulating, and the other two arguments tell it how the string is to be modified. Interesting results can be achieved by treating the third argument as the string to manipulate. Consider this code:

s := "abcde"
write(map("01234", "43201", s))

What does this code example do? The transformation is: "4" should be mapped to "a", "3" to "b", "2" to "c", "0" to "d", and "1" to "e". When this mapping is applied to "01234", we get "edcab" - a permutation of the string s! It is exactly the permutation that
is suggested by the first two arguments of \texttt{map()}. To arrange this sort of permutation, all three strings must be the same size, and there must be no repeated letters in the second string.

![Figure 4-2: Permuting a string with the map() function](image)

Here is an example: In the USA, dates are represented with the month coming first, as in 12/25/1998, but in many other places the day comes first: 25/12/1998. This conversion is, of course, just a permutation; we can do this with a simple map:

\begin{verbatim}
map("Mm/Dd/XxYy", "Dd/Mm/XxYy", date)
\end{verbatim}

Here is another example. Unicon has a built-in random facility, the \texttt{?} operator. Applied to a string or cset, it returns a random character from the argument; applied to a structure, a random member of that structure; and applied to an integer, a random integer between 1 and that number. This is a very useful feature and allows us to write programs that shuffle cards or run simulations of things like rolling dice.

By default in Unicon, the random sequence generated by \texttt{?} is different for each run of the program. This is one of the few areas where Unicon is deliberately different from Icon, which uses the same seed each run by default. Icon’s semantics is good when debugging, because we want the program to behave predictably while it is broken! However, in most applications that use random numbers, such as games, different runs of the program should create different numbers. The random number seed is keyword \texttt{&random}. It can be assigned a value at the start of \texttt{main()} in order to get Icon-style repeatability. Here’s how to assign it a number based on the current date and time. Unicon’s default semantics do something similar.

\begin{verbatim}
&random := map("sSmMhH", "Hh:Mm:Ss", &clock) +
            map("YyXxMmDd", "YyXx/Mm/Dd", &date)
\end{verbatim}

The calls to \texttt{map()} remove punctuation characters from the fixed-format strings produced by \texttt{&clock} and \texttt{&date}. The resulting strings of digits are converted to integers, added, and stored as a seed in \texttt{&random}. Now every time the program is run, the random number facility will be initialized with a different number.
4.8 Simulation

A Galton Box demonstrates how balls falling through a lattice of pegs will end up distributed binomially. A simulation of a Galton box combines several of the techniques described previously. Figure 4-3 is an illustration of the program’s screen.

![Figure 4-3: A Galton Box Simulation](image)

The simulation’s output window is a good example of Unicon’s high-level graphics facilities. Graphics is a broad area, discussed in Chapter 7 of this book; the on-line references or the Icon graphics book (Griswold, Jeffery, Townsend 1998) contain substantial additional details. Graphics are part of the system interface. Some of the graphics functions used in this example include:

- **FillArc(x,y,width,height)** fills an ellipse defined by a bounding rectangle. The shape is filled by the current foreground color and/or fill pattern. The height defaults to be the same as the width, producing a circle. Given additional arguments, `FillArc()` fills parts of an ellipse similar to pieces of a pie in shape.

- **WAttrib("attr")** or **WAttrib("attr=value")**, the generic getter/setter for window attributes. In this case the attributes `fg` (foreground color) and `drawop` (raster drawing operation) are set to various colors and reversible output.

- **Window("attr=value", ...)** opens a window with characteristics specified by string attribute values. The `WDelay(t)` function waits until `t` milliseconds have passed. The
WDone() function waits for the user to dismiss the output window by pressing "q" and then terminates the program and closes the window.

Listing 4-2 contains the code for a simplified version of the simulation. A couple elements of the image above are omitted in order to make the example easy to follow. (Both this program and the one that created the screen shot above are included on the book’s website, http://unicon.org/book/)

Listing 4-2 A Simple Galton Box Simulation

link graphics

global pegsize, height, width, pegsize2

procedure main(args)
local n, steps := 10
  pegsize := 10
  pegsize2 := pegsize * 2
  n := integer(args[1]) | 100
setup_window(steps)
every 1 to n do galton(steps)
WDone()
end

procedure setup_window(n)
local max, xpos, ypos, i, j
  # Draw the n levels of pegs
  # Pegboard size is 2n-1 square
  # Expected max value of histogram is (n, n/2)/2^n
  # ... approximate with something simpler?
  max := n*n/pegsize
  width := (2*n+1)*pegsize
  height := width + n*n/2*pegsize
Window("size=" || width || "," || height, "fg=grayish-white")
WAttrib("fg=dark-grey")
every i := 1 to n do {
  ypos := i * pegsize2
  xpos := width/2 - (i - 1) * pegsize - pegsize/2
  every j := 1 to i do {
    FillArc(xpos, ypos, pegsize, pegsize)
    xpos += pegsize2
  }
}
  # Set up drawing mode to draw the falling balls
WAttrib("fg=black")
4.8. SIMULATION

```
WAttrib("drawop=reverse")
end

# Do it!
procedure galton(n)
local xpos, ypos, oldx, oldy
  xpos := oldx := width/2 - pegsize/2
  ypos := oldy := pegsize
  # For every ball...
  every 1 to n do {
    if ?2 = 1 then xpos -:= pegsize
    else xpos +:= pegsize
    ypos +:= pegsize2
    animate(oldx, oldy, xpos, ypos)
    oldx := xpos; oldy := ypos
  }
  # Now the ball falls to the floor
  animate(xpos, ypos, xpos, ypos + 40)
  animate(xpos, ypos+40, xpos, ypos + 200)
  # Record this ball
  draw_ball(xpos)
end

procedure animate(xfrom, yfrom, xto, yto)
  animate_actual(xfrom, yfrom, xto, yfrom, 4)
  animate_actual(xto, yfrom, xto, yto, 10)
end

# Drawing op is already set to "reverse", and fg colour is black.
procedure animate_actual(xfrom, yfrom, xto, yto, steps)
local x := xfrom, y := yfrom, xstep, ystep, i, lastx, lasty
  xstep := (xto - xfrom)/steps
  ystep := (yto - yfrom)/steps
  every i := 1 to steps do {
    lastx := x; lasty := y
    FillArc(x, y, pegsize, pegsize)
    WDelay(1)
    FillArc(x, y, pegsize, pegsize)
    x +:= xstep; y +:= ystep
  }
end

procedure draw_ball(x)
static ballcounts
initial ballcounts := table(0)
  ballcounts[x] +:= 1
  FillArc(x, height-ballcounts[x]*pegsize, pegsize, pegsize)
end
```
Summary

Unicon is particularly powerful when language features are combined. The ability to combine features in interesting ways is the result of its novel expression semantics. Co-expressions add substantial value to the concept of generators, although most programs use them only sparingly. They fit well into a philosophy that says that simple things should be easy to do...and complex things should be easy to do as well.
Chapter 5

The System Interface

The system interface is Unicon’s connection to the outside world, defining input/output interactions with the operating system. This chapter shows how to

- Manipulate files, directories, and access permissions
- Launch and interact with other programs
- Handle abnormal events that would otherwise terminate your program
- Write Internet client and server applications.

5.1 The Role of the System Interface

Unicon’s predecessor Icon is highly portable; it runs on everything from mainframes to Unix machines to Amigas and Macs. This platform independence is both a virtue and a limitation. Icon takes a greatest common denominator approach to the system interface. Icon programs run with no source modifications across platforms, but with little access to the underlying system. Icon historically could not be used easily for many tasks such as system administration or client/server programming. Both the Icon graphics facilities, and now the Unicon system interface, “raise the bar” of what portable facilities programmers can expect to be provided by their programming language, at the cost of making it more difficult to port the language to new platforms.

The interface described in this chapter relies on underlying standards including ANSI C’s standard library, and the IEEE Portable Application Standards Committee’s POSIX operating system standard (http://www.pasc.org). Unicon relies on standards, but is simpler and higher level. It is also less platform-specific than the POSIX standard. The goal was to define facilities that can be implemented to a great extent on all modern operating systems. Non-POSIX Unicon implementations may provide a subset of the functionality described in this chapter, but important facilities such as TCP/IP Internet communications are ubiquitous and warrant inclusion in the language definition. So far the complete
Unicon system interface is implemented for Linux, Solaris, and Windows; the challenge to port these facilities to all platforms on which they are relevant and useful now rests with Unicon’s user community.

5.2 Files and Directories

The file type is used for any connection between a program and an external piece of hardware. In reality, a file is a reference to resources allocated by the operating system for the purpose of input or output. Different kinds of files support different operations, but most files support the basic functions given in this section.

Files are commonly used to manipulate named repositories of data on a storage device. The contents of files exist independent of the program that creates them, and persist after that program finishes. To read data from a file or save data to a file, the functions read() and write() are often used. These functions by default use special files denoted by the keywords &input and &output, respectively. There is a third file keyword, &errout, that refers to the location to which the program should write any error messages. Unless the files were redirected, they refer to the keyboard and the display. If you pass read() or write() a value of type file as an argument, the operation is performed on that file. The function open() creates a value of type file:

\[
f := \text{open("myfile.txt", "w")}
\]
\[
\text{write}(f, \"This is my text file.\")
\]

The open() function takes two parameters: a file name and a mode. The default mode is "r" for reading; the example above uses mode "w" for writing. Other modes denote other kinds of system interfaces. They are described in later sections.

The read() function reads and returns a line of text, removing the line terminator(s). Function write() similarly adds a line terminator after writing its arguments. Another way to read lines is via the generate operator, unary !. The expression ![f] generates the lines of file f, so every put(L, ![f]) puts the lines of f into list L.

On systems with multi-character line terminators, appending an extra letter to the mode parameter of open() indicates whether newlines are to be translated (mode "t") or untranslated (mode "u"). Text files should be translated, while binary files should not. The default is to translate newlines to and from operating system format.

Besides read() and write(), which always process a single line of text, the functions reads(f, i) and writes(f, s, ...) read (up to i characters) and write strings to a file. These functions are not line-oriented and do no newline processing of their own, although they still observe the translation mode on systems that use one.

When operations on a file are complete, close the file by calling close(f). The only exceptions are the standard files, &input, &output, and &errout; since you didn’t open them, don’t close them. For the rest, most operating systems have a limit on the number
of files that they can have open at any one time, so not closing your files can cause your program to fail in strange ways if you use a lot of files.

**Directories**

A directory is a special file that contains a collection of named files. Directories can contain other directories to form a hierarchical structure. The `chdir()` function returns the current working directory as an absolute path name. Called with a string argument, the `chdir(dirname)` function sets the current working directory to `dirname`. The call `open(dirname)` opens a directory to read its contents. Directories can only be opened for reading, not for writing. Every `read()` from a directory returns the name of one file. Directory entries are not guaranteed to be in any order. The expression `every write(!open("."))` writes the names of the files in the current directory, one per line. It is not good practice to call an `open()` that you don’t `close()`.

The `mkdir(s)` function creates a directory. An optional second parameter specifies access permissions for the directory; controlling file ownership and access is discussed below. Files or directories can be renamed with `rename(s1,s2)`. Renaming does not physically move the file, so if `s1` and `s2` denote locations on different hardware devices or file systems then `rename()` will fail, and you will need to “copy and then delete” the file. Individual files or directories are removed with `remove(s)`. Only empty directories may be removed. To remove an entire directory including its contents:

```pascal
procedure deldir(s)
  f := open(s)
  every remove( s || "/" || ("." ~== (".." ~== f)))
  close(f)
  remove(s)
end
```

How would you change this function to delete subdirectories? You might be able to devise a brute force approach using what you know, but what you really need is more information about a file, such as whether it is a directory or not.

**Obtaining file information**

*Metadata* is information about the file itself, as opposed to information stored in the file. Metadata includes the owner of the file, its size, user access rights, and so forth. This information is produced by the `stat()` system call. Its argument is the name of a file or (on UNIX systems only) an open file. The `stat()` function returns a record with the information about the file. Here is a subset of `ls`, a UNIX program that reads a directory and lists information about its files. Keyword `&errortext` contains information about the most recent error that resulted in an expression failure; it is written if opening the directory
fails. This version of ls only works correctly if its arguments are the names of directories. How would you modify it, using stat(), to take either ordinary file names or directory names as command line arguments?

```c
link printf
procedure main(args)
    every name := !args do {
        f := open(name) | stop(&errortext, name)
        L := list()
        while line := read(f) do
            push(L, line)
        every write(format(stat(n := !sort(L)), n))
    }
end
procedure format(p, name)
    s := sprintf("%-7d %-5d %s %-4d %-9d %-9d %-8d %s %s",
        p.ino, p.blocks, p.mode, p.nlink, p.uid, p.gid, p.size,
        ctime(p.mtime)[5:17], name)
    if p.mode[1] == "l" then
        s ||:= " -> " || (p.symlink)
    return s
end
```

The record returned by stat() contains many fields. Not all file systems support all of these fields. Two of the most important portable fields are size, the file size in bytes, and mtime, the file’s last modified time, an integer that is converted into a human readable string format by ctime(). Another important field is mode, a string that indicates the file’s type and access permissions. Its first letter (mode[1]) is "," for normal files, "d" for directories, and some file systems support additional types. The other characters of the mode string are platform dependent. On UNIX there are nine letters to encode read, write, and execute permissions for user, group, and world, in the format: "rwxrwxrwx". On a classic Windows FAT file system, there is only "rwa" to indicate the status of hidden, read-only, and archive bits (if it is set, the system bit is indicated in mode[1]).

Some file systems support duplicate directory entries called links that refer to the same file. In the record returned by stat(), a link is indicated by a mode[1] value of "l". In addition, field nlinks ("number of links") will be > 1 and/or field symlink may be the string filename of which this file is an alias. Appendix E includes information on each platform’s support for the mode field, as well as stat()’s other fields.

**Controlling file ownership and access**

The previous section shows how different platforms’ file systems vary in their support for the concepts of file ownership and access. If the system supports ownership, the user and
5.3. PROGRAMS AND PROCESS CONTROL

group that own a file are changed by calling \texttt{chown(fname, user, group)}. The \texttt{chown()} function only succeeds for certain users, such as the super user. User and group may be string names, or integer user identity codes on some platforms.

File access rights are changed with \texttt{chmod(fname, mode)}. The \texttt{chmod()} function only succeeds for the owner of a given file. The \texttt{mode} is a nine-letter string similar to \texttt{stat()}’s mode field, or an octal encoding of that information (see Appendix E).

Another piece of information about files is called the \textit{umask}. This is a variable that tells the system what access rights any newly created files or directories should have. The function call \texttt{umask(“rwxr-xr-x“)} tells the system that newly created directories should have a permission of \texttt{“rwxr-xr-x”} and files should have permissions of \texttt{“rw-r--r--“}. The \texttt{mkdir(s, mode)} function takes an optional mode parameter, which can override the umask for for newly created directories. Ordinary files are never given execute permission by the system, it must be set explicitly with \texttt{chmod()}.

\textbf{File locks}

Files can be locked while a program is updating some information. If the contents of the file are in an inconsistent state, other programs may be prevented from reading (or especially writing) the file. Programs can cooperate by using file locks:

\texttt{flock(filename, “x“)}

The first call to \texttt{flock()} creates a lock, and subsequent calls by other programs will block, waiting till the writing program releases its lock. The flag \texttt{“x“} represents an \textit{exclusive} lock, which should be used while writing; this means no other process can be granted a lock. For reading, \texttt{“s“} should be used to create a shared lock so that other programs that are also just reading can do so. In this way you can enforce the behavior that only one process may open the file for writing, and all others will be locked out; but many processes can concurrently open the file for reading.

\section{5.3 Programs and Process Control}

Unicon’s system interface is similar but higher level than the POSIX C interface. An include file \texttt{posix.icn} defines constants used by some functions. Include files are special code, consisting mainly of defined symbols, intended to be textually copied into other code files. They are handled by the preprocessor, described in Appendix A. To include \texttt{posix.icn} in a program, add this line at the top of your program:

\texttt{${include "posix.icn"}}

When a system call fails, the integer keyword \texttt{&errno} indicates the error that occurred. As seen earlier, a human-readable string is also available in \texttt{&errortext}. Error codes (such
as EPERM, or EPIPE) are defined in posix.icn; &errno can be compared against constants like ENOENT. In general, however, human readers will prefer to decipher &errortext.

In the discussion to follow, a program is the code, while a process is such a program in execution. This distinction is not usually important, but for network applications it matters, since the same program can run in multiple processes, and a process can change the program that it is running.

Signals

A signal is an asynchronous message sent to a process either by the system (usually as a result of an illegal operation like a floating point error) or by another process. A program has two options to deal with a signal: it can allow the system to handle it in the default manner (which may include termination of the process) or it can register a function, called a signal handler, to be run when that signal is delivered.

Signals are trapped or ignored with the trap(s, p) function. Argument s is the string name of the signal. The signal names vary by platform; see Appendix E. You can trap any signal on any machine; if it is not defined it will be ignored. For example, Linux systems don’t have a SIGLOST. Trapping that signal has no effect when a program runs on Linux. The trap() function’s second argument is the procedure to call when the signal is received. The previous signal handler is returned from trap() so it can be restored by a subsequent call to trap(). The signal handler defaults to the default provided by the system. For instance, SIGHUP is ignored by default but SIGFPE will cause the program to terminate.

Here is an example that handles a SIGFPE (floating point exception) by printing out a message and then runs the system default handler:

global oldhandler

... trap("SIGFPE", sig_ignore)
oldhandler := signal("SIGSEGV", handler)
...
# restore the old handler
trap("SIGSEGV", oldhandler)
end

procedure sig_ignore(s); end
procedure handler(s)
    write(&errout, "Got signal ", s)
    (oldhandler)(s)
    # propagate the signal
end
Launching programs

Many applications execute other programs and read their results. In many cases, the best way to do this is to call `open()` with mode "p" (pipe) to launch a command. In mode "p" the string argument to `open()` is not a filename, it is an entire command string. Piped commands opened for reading (mode "p" or "pr") let your program read the command’s standard output, while piped commands open for writing (mode "pw") allow your program to write to the command’s standard input.

The more general function `system(x,f1,f2,f3,mode)` runs an external command (argument x) with several options. If x is a list, x[1] is the command to execute and the remaining list elements are its command line arguments. If x is a string, it is parsed into arguments separated by spaces. Arguments with spaces in them may be escaped using double quotes. A program that calls `system()` normally waits for the launched program to complete before continuing, and `system()` returns the integer status of the completed command. If s ends in an ampersand (&) or the optional mode argument is 1 or "nowait", `system()` does not wait for the command to complete, but instead launches the command in the background and returns an integer process id. The `system()` function takes three optional file arguments that specify redirected standard input, output, and error files for the launched program.

Using file redirection and pipes

One common scenario is for a program to run another program but with the input and output redirected to files. On command-line systems like the Unix shells or the MS-DOS command prompt, you may have used redirection:

```
prog < file1
```

File redirection characters and other platform-dependent operations are supported in the command string passed to `system()`, as in the following `system()` call:

```
system("prog < file1")
```

Pipes to and from the current program are nicely handled by the `open()` function, but sometimes the input of one program needs to be connected to the output of another program. You may have seen uses like this:

```
prog1 | prog2
```

The `pipe()` function returns a pair of open files in a list, with the property that anything written to the second file will appear on the first. Here’s how to hook up a pipe between two programs:
L := pipe() | stop("Couldn’t get pipe: ", &errortext)
 system("prog1 &", , L[2])
 system("prog2 &", L[1])
 close(L[1])
 close(L[2])

Process information

The integer process identity can be obtained with getpid(). The user id of the process can be obtained with getuid() if the platform supports it. Calls to obtain additional information such as group identity on some platforms are described in Appendix E.

A parent process may want to be notified when any of its children quit (or change status). This status can be obtained with the function wait(). When a child process changes state from “running” to either “exited” or “terminated” (and optionally “stopped”), wait() returns a string of the form

    pid:status:arg:core

The ":core" will only be present if the system created a core file for the process. The status can be any of “exited”, “terminated” or “stopped”. The arg field is either: a) the exit status of the program if it exited; or b) the signal name if it was terminated. Typically wait() will be used in the handler for the SIGCHLD signal which is sent to a process when any of its children changes state.

The arguments to wait() are the pid of the process to wait for and any options. The default for pid is to wait for all children. The options may be either "n", meaning wait() should not wait for children to block but should return immediately with any status that’s available, or "u", meaning that any processes that stopped should also be reported. These options may be combined by using "nu".

The select() system call

Some programs need to be able to read data from more than one source. For example, a program may have to handle network traffic and also respond to the keyboard. The problem with using read() is that if no input is available, the program will block and will not be able to handle the other stream that may in fact have input waiting on it. To handle this situation, you can use the function select(x1,x2,...i). The select() function tells the system which files you are interested in reading from, and when input becomes available on any of those sources, the program will be notified. The select() function takes files or lists of files as its arguments, and returns a list of all files on which input is waiting. If an integer argument is supplied, it is a timeout that gives the maximum milliseconds to wait before input is available. If the timeout expires, an empty list is returned. If no timeout is given, the program waits indefinitely for input on one of the files.
5.4. NETWORKING

```plaintext
while *(L := select(f1, f2, f3, timeout)) = 0 do
  handle_timeout()
  (&errno = 0) | stop("Select failed: ", &errortext)
every f := !L do {
  # Dispatch reads pending on f
  ...
}
```

When using `select()` to process input from multiple files, you may need to pay some attention to avoid blocking on any one of your files. For example, the function `read()` waits until an entire line has been typed and then returns the whole line. Consider this code, which waits for input from either a file (or network connection) or a window designated by keyword `&window`:

```plaintext
while L := select(f, &window) do
  if !L === f then c := read(f)
```

Just because `select()` has returned doesn’t mean an entire line is available; `select()` only guarantees that at least one character is available. The command shell log application in Chapter 14 shows the usage of `select()`. Another primary application area for `select()` is network programming, described later in this chapter. For network connections, the function `reads(f, i)` will return as soon as it has some input characters available, rather than waiting for its maximum string size of `i`. But if no input is available, `reads()` blocks.

### Non-blocking input and the `ready()` function

The function `ready(f, i)` is like `reads(f, i)` except that it is non-blocking, that is, it returns immediately with up to `i` bytes if they are available, but it does not wait around. It is ideal for use with `select()` and in situations where a server or client needs to interact with multiple remote connections.

### 5.4 Networking

Unicon provides a very high-level interface to Internet communications. Applications with custom communications use one of the major Internet applications protocols, TCP and UDP. An higher level interface to several popular Internet protocols such as HTTP and POP is provided by means of Unicon’s messaging facilities.

### TCP

A TCP connection is a lot like a phone call: to make a connection you need to know the address of the other end, just like a phone number. For TCP, you need to know the name of
the machine to connect to, and an address on that machine, called a port. A server listens for connections to a port; a client sends requests to a port. Also, there are two kinds of ports, called "Internet Domain" and "Unix Domain." The distinction is beyond the scope of this book; we will just mention that Internet Domain ports are numbers, and Unix Domain ports look like files. Also, a connection to a Unix domain port can only be made from the same machine, so we will not consider the Unix domain further here.

A call to open() with mode "n" (network) requests a network connection. The first argument to open() is the network address, a host:port pair for Internet domain connections, and a filename for Unix domain sockets. If the address contains no host name and therefore starts with ":", the socket is opened on the same machine. The value returned by open() is a file that can be used in select() and related system functions, as well as normal reading and writing.

A client uses mode "n" with open() to open a connection to a TCP server. Here is a simple version of the Internet "finger" program:

```pascal
procedure main(argv)
  local fserv := getserv("finger") |
    stop("Couldn’t get service: ", &errortext)
  name := argv[1]
  host := ""
  argv[1] ? {
    name := tab(find("@")) & ="@" & host := tab(0)
  }
  if *host > 0 then write("[", host, "]")
  f := open(host || ":" || fserv.port, "n") |
    stop("Couldn’t open connection: ", &errortext)

  write(f, name) | stop(" Couldn’t write: ", &errortext)
  while write(read(f))
end
```

Notice the use of getserv(). The posix_servent record it returns includes fields for the name, aliases, port, and protocol used by the Internet service indicated in getserv()’s argument. The Internet protocols specify the ports to be used for various services; for instance, email uses port 25. Instead of having to remember port numbers or hard-coding them in our program, we can just use the name of the service and have getserv() translate that into the port number and protocol we need to use.

To write a server, all we need to do is add "a" (accept) to the mode after the "n" in open(). Here is a simple TCP server that listens on port 1888:

```pascal
procedure main()
  while f := open(":1888", "na") do {
    system("myserverd -servicerequest ", f, f)
```
The call `open(":1888", "na")` blocks until a client connects. The returned value is a file that represents the network connection. This example server responds to requests by launching a separate process to handle each request. The network connection is passed to `myserverd` as its standard input and output, so that process had better be expecting a socket on its standard I/O files, and handle it appropriately. This works on UNIX; on other platforms a different approach is needed.

Launching a separate process to handle requests is standard operating procedure for many Internet servers, but besides the portability concerns, it uses a lot of memory and CPU time. Many servers can be implemented in a single process. Chapter 15 includes an example of such a server. Mode "na" is less than ideal for one-process servers: it only supports one connection at a time. When waiting for a new connection, the process is not doing any computation, and when servicing a connection, the program is not listening for any other connection requests. Unless each connection is of short duration, the server will appear to be down, or appear to be unacceptably slow, to anyone trying to connect while an existing request is being processed.

### Determining IP numbers

Many programs need the IP number of the machine they are talking to. Given a network connection `f`, `image(f)` will show the IP address and port of the client machine that is connected (this is sometimes called the peername).

Some programs need to know their own IP number, but each machine can have several IP numbers, one for each kind of physical network hardware in operation. To obtain a list of local IP numbers, a program can read the output of `/sbin/ifconfig` (UNIX) or `ipconfig` (Windows). To find the IP number used for a particular network connection `n`, on some platforms you can call `gethost(n)`, which returns a string with the IP number and port used by the local machine for a given connection.

If you do determine your IP number in one of these ways, it is usually not the number seen by the world, because most devices are connecting through some form of network address translation. To see the number that the world sees, you have to connect to someone else and ask them to tell you what IP number they see you at.
Non-blocking network opens

Servers need to never block. The call \texttt{open(":port","nl")} creates a \texttt{listener} on the specified port, without waiting around for someone to actually connect to it. The network file returned from \texttt{open()} is not open for reading or writing, so it is not good for much...yet. About the only thing you can do with such a file is include it (along with any other network connections you have going) as an argument in a call to \texttt{select()}. If a listener matches a current connection request, \texttt{select()} converts it into a regular network connection as per mode "na".

In addition to non-blocking servers’ listener connections, in the real-world clients need a way to do an almost non-blocking connection as well. TCP connections over long distances take a highly variable amount of time, but most clients do not want to “freeze” for a couple of minutes while the connection attempt times out. The network client versions of \texttt{open()} allows an optional third parameter to supply it with a timeout value, in milliseconds.

UDP

UDP is another protocol used on the Internet. TCP is like a phone call: all messages you send on the connection are guaranteed to arrive in the same order they were sent. UDP on the other hand is more like the postal service, where messages are not guaranteed to reach their destination and may not arrive in the same order they were sent in. Messages sent via UDP are called datagrams. It’s a lot cheaper (faster) to send UDP messages than TCP, though, especially if you are sending them across the Internet rather than to a local machine. Sending a postcard is usually cheaper than a long distance phone call!

UDP datagrams can be sent either with an \texttt{open()/writes()} pair, or with \texttt{send()}. Typically a server sends/receives on the same socket so it will use \texttt{open()} with \texttt{read()} and \texttt{write()}. A client that only sends one or two datagrams uses \texttt{send()/receive()}.

The following example provides a service called “rdate” that allows a program to ask a remote host what time it has. The server waits for request datagrams and replies with the date and time. The "u" flag added to the mode in the call to \texttt{open()} signifies that a UDP connection should be used. The function \texttt{receive()} waits for a datagram to arrive, and then it constructs a record having the address the message came from and the message in it. The server uses the address to send the reply.

```c
f := open("":1025", "nua")
while r := receive(f) do {
    # Process the request in r.msg
    ...
    send(r.addr, reply)
}
```
The record returned by `receive()` has two fields: the `addr` field contains the address of the sender in "host:port" form, and the `msg` field contains the message.

To write a UDP client, use mode "nu" Since UDP is not reliable, the `receive()` is guarded with a `select()`; otherwise, the program might hang forever if the reply is lost. The timeout of five seconds in the call to `select()` is arbitrary and might not be long enough on a congested network or to access a very remote host. Notice the second argument to `getserv()`; it restricts the search for Internet service information to a particular network protocol, in this case UDP.

```
procedure main(args)
  (*args = 1) | stop("Usage: rdate host")
  host := args[1]
  s := getserv("daytime", "udp")
  f := open(host||":"||s.port, "nu") |
          stop("Open failed: ", &errortext)
  writes(f, " ")
  if *select(f, 5000) = 0 then
    stop("Connection timed out.")
  r := receive(f)
  write("Time on ", host, " is ", r.msg)
end
```

From these examples you can see that it is relatively easy to write programs that use Internet communication. But TCP and UDP are very general, somewhat low-level protocols; most programs employ a higher-level communication protocol, either by defining their own, or using a standard protocol. If you need to define your own Internet protocol, you can do it on top of TCP or UDP; if your program needs to use a standard Internet protocol, you should check first to see if the protocol is built-in to the language as part of the messaging facilities, described in the next section.

### 5.5 Messaging Facilities

Unicon’s messaging facilities provide higher level access to many popular Internet protocols. A call to `open()` using mode "m" establishes a messaging connection. The filename argument to a messaging connection is a URI (Uniform Resource Indicator) that specifies the protocol, host machine, and resource to read or write. The protocols implemented thusfar are HTTP, Finger, SMTP, and POP. Extra arguments to `open()` are used to send headers defined by the protocol. For example, the call

```
open("mailto:unicon-group", "m", "Reply To: jeffery@cs.uidaho.edu")
```

supplies a Reply To field as a third parameter to `open()` on an SMTP connection.

Header fields from the server’s response to a connection are read by subscribing the message connection value with a string header name; an example is in the next section.
HTTP

HTTP is used to read or write to Web servers; the content read or written typically consists of HTML text. The following program, `httpget.icn`, fetches a remote file specified by a URI on the command line, and saves it as a local file. The Icon Program Library module `basename` is used to extract the filename from the URI.

```
link basename

procedure main(argv)
    f1 := open(argv[1],"m")
    f2 := open(basename(argv[1]),"w")
    while write(f2, read(f1))
end
```

This example retrieves the actual data for a successful HTTP request; for a failed request the connection returns no data, appearing to be an empty file. Programs can check the HTTP status code in order to determine the nature of the problem. Status codes and other metadata from HTTP header lines are inspected by subscripting the connection with the desired header. For example, in the previous program, checking `f1["Status-Code"]` would allow us to detect HTTP errors, and `f1["Location"]` would allow us to find the new location of the file, if the HTTP server had informed us that the file had moved. You can retrieve this status information on a remote file without retrieving the file itself. If you open a URI with mode "ms" instead of "m", an HTTP request for header information is made, but no data is retrieved.

SMTP and POP

SMTP is used to send a mail message. Mail is delivered via an SMTP server machine on which the user must have a valid account. These default to the current user on the current host machine. Two environment variables UNICON_SMTPSERVER and UNICON_USERADDRESS can be set to override the default behavior.

POP is used to read mail messages. POP is the least file-like of these protocols. A POP connection feels like a list of strings, each string containing an entire mail message, rather than a simple sequence of bytes. Function `read()` and operator `!` produce entire messages which in general contain many newline characters. POP messages may be removed by either `delete()` or `pop()`; messages are buffered in such a way that message removal on the server occurs when the connection is explicitly and successfully closed.

Here’s a simple program that illustrates the use of msesaging to get email from a POP server. It gets messages from the server without deleting them and, for every message, prints out who the message is from as well as the subject line.
procedure main(argv)
    user := argv[1] | getenv("USER") | stop("no user")
    passwd := argv[2] | getenv("PASSWD") | stop("no password")
    server := argv[3] | getenv("MAILHOST") | "mailhost"
    conn := open("pop://"||user||":"||passwd||"@"||server, "m") |
        stop("couldn’t connect to the POP server ", server)
    every msg := !conn do msg ?
        while line := tab(find("\n")) do
            if =("From: " | "Subject: ") then write(line)
        close(conn)
end

You should improve the password handling in this program if you use it! Chapter 14 includes another example use of Unicon’s POP messaging facilities: a spam filter.

5.6 Tasks

A task is an executing program within the Unicon virtual machine. A single task called the root is created when the interpreter starts execution. Additional tasks are created dynamically as needed.

The tasking facilities allow Unicon programs to load, execute, communicate with, and control one another, all within a single instantiation of the Unicon interpreter.

Although Unicon programs can use the tasking facilities to load and execute any number of other programs within the same interpreter space, the tasking facilities do not introduce a concurrent programming construct nor do they include special support for multiprocessor hardware. Their domain is that of high-level language support for programs that benefit from or require a tighter coupling than that provided by inter-process communication; that is, programs that access each other’s state extensively.

Co-expressions provide the tasking facilities’ program execution model, and co-expression activation serves as the communication mechanism. The extensions are general enough to be useful in a wide variety of contexts. For example, programs that use the multi-tasking interface can communicate directly without resorting to external files or pipes.

At the language level, the tasking facilities include several built-in functions and keywords, but no new types, declarations, or control structures. Several existing functions have been extended to offer additional support for the multi-tasking environment. Separate memory allocation regions are established for each task.

Preliminary terminology

Before describing the task model, a few definitions are needed. These definitions pertain to regions of memory referenced by programs during execution.
A name space is a mapping from a set of program source-code identifiers to a set of associated memory locations [Abelson85]. Icon programs have a global name space shared across the entire program and various name spaces associated with procedures. Procedures each have a static name space consisting of memory locations shared by all invocations of the procedure and local name spaces private to each individual invocation of the procedure. When a co-expression is created, a new local name space is allocated for the currently executing procedure, and the current values of the local variables are copied into the new name space for subsequent use by the co-expression.

An Icon program has an associated program state consisting of the memory associated with global and static name spaces, keywords, and dynamic memory regions. Similarly, a co-expression has an associated co-expression state consisting of an evaluation stack that contains the memory used to implement one or more local name spaces. Co-expressions in an Icon program share access to the program state and can use it to communicate.

**Tasks as extended co-expressions**

A task consists of a main co-expression and zero or more child co-expressions that share a program state consisting of the global and static name spaces, keywords, and dynamic memory regions. At the source-language level, tasks are loaded, referenced, and activated solely in terms of one of their member co-expressions; the task itself is implicit. Co-expressions share access to the program state and can use it to communicate. Unicon provides the task model as a mechanism for multi-tasking, but does not predefine the policy; matters such as the scheduling algorithm used and whether multi-tasking is co-operative or pre-emptive are programmable at the user level in terms of co-expression activations.

**Task creation**

A task can create other tasks. The function

\[
\text{load}(s, L, f1, f2, f3, i1, i2, i3)
\]

loads an icode file [Gris86] specified by the file name s, creates a task for it and returns a co-expression corresponding to the invocation of the procedure main(L) in the loaded icode file. L defaults to the empty list. Unlike conventional Icon command-line argument lists, the argument list passed to load() can contain values of any type, such as procedures, lists, and tables in the calling task.

The task being loaded is termed the child task, while the task calling load() is termed the parent. The collection of all tasks forms a tree of parent-child relationships.

f1, f2, and f3 are file arguments to use as &input, &output, and &error in the loaded task; &input, &output, and &error default to those of the loading task. i1, i2, and i3 are three integer arguments that supply initial region sizes for the task’s block, string,
and stack memory areas, respectively. i1 and i2 default to 65000, while i3 defaults to 20000 (the defaults may be changed by the environment variables BLKSIZE, STRSIZE, and MSTKSIZEx).

### Running other programs

A co-expression created by `load()` is activated like any other co-expression. When activated with the `@` operator, the child task begins executing its main procedure. Unless it suspends or activates `&source`, the child task runs to completion, after which control is returned to the parent. Chapter 5 presents an alternative means of executing a child with which the parent retains control over the child as it executes.

This default behavior is illustrated by the program `seqload`, which loads and executes each of its arguments (string names of executable Unicon programs) in turn. Each of the strings passed on the command line and extracted from the list using the element-generation operator, `!`, is passed in turn to `load()`, which reads the code for each argument and creates a task in which to execute the loaded program. The tasks are then executed one-by-one by the co-expression activation operator, `@`. There is nothing special about this example except the semantics of the `load()` function and the independent execution environment (separate global variables, heaps, and so forth), that `load()` provides to each task.

```plaintext
# seqload.icn
procedure main(arguments)
    every @load(arguments)
end
```

For example, if three Icon programs whose executable files are named translate, assemble, and link are to be run in succession, the command

```
seqload translate assemble link
```

executes the three programs without reloading the interpreter for each program.

### Data access

Although tasks have separate program states, they reside in the same address space and can share data; values can be transmitted from task to task through `main()`’s argument list, through co-expression activation, or by use of event monitoring facilities described in Chapter 10. In the following pair of programs, the parent receives a list value from the child and writes its elements out in reverse order.

```plaintext
# parent.icn
procedure main()
    L := @ load("child")
```
while write(pull(L))
end

# child.icn
procedure main()
  L := []
  while put(L, read())
    return L
end

Access Through Task Argument Lists

The following program takes its first argument to be an Icon program to load and execute as a child, sorts its remaining arguments, and supplies them to the child program as its command-line arguments (\texttt{pop()} and \texttt{sort()} are Icon built-in functions that extract the first list element and sort elements, respectively):

```icon
procedure main(arguments)
  @load(pop(arguments), sort(arguments))
end
```

Argument lists allow more sophisticated data transfers; the \texttt{seqload} example presented earlier can be extended to transmit arbitrary structures between programs using argument lists in the following manner. As in \texttt{seqload}, each string naming an executable Icon program is passed into \texttt{load()}, and the resulting task is activated to execute the program. In this case, however, any result that is returned by one of the programs is assigned to local variable \texttt{L} and passed to the next program in the list via the second argument to \texttt{load()}.

```icon
# seqload2.icn
procedure main(arguments)
  every program := !arguments do
    L := @load(program, L)
  end
```

The net effect of \texttt{seqload2.icn} is similar to a UNIX pipe, with an important difference: Arbitrary Icon values can be passed from program to program through the argument lists. This capability is more interesting in substantial multipass tools such as compilers, where full data structures can be passed along from tool to tool instead of writing out text encodings of the structures to a file.

Inter-task Access Functions

Several built-in functions provide inter-task access to program data. These functions are usable in any multi-task Unicon context, but are especially useful in program execution monitoring, discussed in Chapter 9.
5.6. TASKS

For example, the `variable()` function takes a co-expression value as an optional second argument denoting the task from which to fetch the named variable. `variable(s, C)` is useful for assigning to or reading from another task’s variables. In the following `seqload` example, the parent initializes each child’s `Parent` global variable (if there is one) to refer to the parent’s `&main` co-expression. A child task can then use `Parent` to determine whether it is being run standalone or under a parent task. The `variable()` function is useful in inspecting values, especially at intermediate points during the monitored execution of a TP.

```unicon
# seqload3.icn
procedure main(arguments)
    every arg := !arguments do {
        Task := load(arg)
        variable("Parent", Task) := &main @Task
    }
end
```

In addition to extending existing functions for monitoring, several new functions have been added. The use of these monitoring functions are illustrated in many example monitors in Unicon’s `ipl/mprogs` directory. Some of the intertask access functions are listed in Figure 5-1. In these functions, parameter `C` refers to a co-expression that may be from a task other than the one being executed. Functions that `generate` values can produce more than one result from a given call.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cofail(C)</code></td>
<td>transmit failure to <code>C</code>.</td>
</tr>
<tr>
<td><code>fieldname(r)</code></td>
<td>generate fieldnames of record <code>r</code>.</td>
</tr>
<tr>
<td><code>globalnames(C)</code></td>
<td>generate the names of <code>C</code>'s global variables.</td>
</tr>
<tr>
<td><code>keyword(s, C)</code></td>
<td>produces keyword <code>s</code> in <code>C</code>. Keywords are special global variables that have special semantics in certain language facilities.</td>
</tr>
<tr>
<td><code>localnames(C,i)</code></td>
<td>generates the names of <code>C</code>'s local variables, <code>i</code> calls up from the current procedure call.</td>
</tr>
<tr>
<td><code>paramnames(C,i)</code></td>
<td>generates the names of <code>C</code>'s parameters.</td>
</tr>
<tr>
<td><code>staticnames(C,i)</code></td>
<td>generates the names of <code>C</code>'s static variables.</td>
</tr>
<tr>
<td><code>structure(C)</code></td>
<td>generates the values in <code>C</code>'s block region, or heap. The heap holds structure types such as lists and tables.</td>
</tr>
<tr>
<td><code>variable(s,C,i)</code></td>
<td>produces variable named <code>s</code>, interpreted <code>i</code> levels up within <code>C</code>'s procedure stack.</td>
</tr>
</tbody>
</table>

Figure 5-1: Unicon interprogram access functions.
Shared icode libraries

Programs that are written to take advantage of the multi-tasking environment gain in space efficiency and modularity. Code sharing is one natural way to achieve space efficiency in a collection of programs. Since procedures are first-class data values in Icon, code sharing can be implemented via data sharing. Programs executing in a single invocation of the interpreter can share code easily if the code is not required to produce side effects on global variables in the calling task’s program state. If side-effects to the calling task’s program state are required, the shared code must generally be written with care to explicitly reference the calling task’s state. Side effects in the client task can also be achieved through the parameters passed in and results obtained by calling the shared procedure.

Loading shared code

Consider a collection of applications that make extensive use of procedures found in the Icon program library (IPL) [Griswold90c]. If those applications are run using MT Icon, the IPL routines need be loaded only once, after which they may be shared.

In order to reference shared code from a loaded task, two additional considerations must be satisfied: the shared code must be loaded, and the client tasks must be able to dynamically link shared routines into their generated code.

Both of these problems can be solved entirely at the source level: In order to introduce a shared Icon procedure into the name space, a global variable of the same name must be declared. Managing the loading of shared libraries is itself a natural task to assign to an Icon procedure that uses a table to map strings to the pointers to the procedures in question.

Code sharing example

The following collection of three programs illustrate one schema that allows code sharing. Other conventions can certainly be devised, and much of the sharing infrastructure presented here can be automatically generated. Program calc.icn consists of a shared library procedure named \texttt{calc()} and a main procedure that exports a reference to \texttt{calc()} for sharing:

```plaintext
# calc.icn
procedure calc(args...)
    # code for calc
    # (may call other routines in calc.icn if there are any)
end

procedure main()
    # initialization code, if any
    return calc
end
```
Note that a module exporting shared procedures can also have global variables (possibly initialized from other command-line arguments). Shared modules can export other values besides procedures using the same mechanism.

The parent task that loads the various shared library clients implements a procedural encapsulation (loadlib() in this example) of an Icon table to store references to shared routines. The parent passes this procedure to clients. Each client calls the procedure for each shared routine. Routines that are already loaded are returned to requesting tasks after a simple Icon table lookup. Whenever a routine is requested that has not been loaded, the load() function is called and the shared library activated.

procedure main(arguments)
    @load("client",put(arguments,loadlib))
end

procedure loadlib(s, C)
    static sharedlib
    initial sharedlib := table()
    sharedlib[s] := @load(s)
    variable(s, C) := sharedlib[s]
end

A client of calc declares a global variable named calc, and assigns its value after inspecting its argument list to find the shared library loader:

global loadlib
global calc
procedure main(arguments)
    if /loadlib then stop("no shared libraries present")
    loadlib("calc", &current)
    # ... remainder of program may call shared calc
end

Sharing procedure collections

The primary deficiency of the previous example is that it requires one shared library procedure per Icon module, that is, separate compilation. In practice it is more convenient to have a collection of related procedures in a given Icon compilation unit. Shared libraries can employ such a mechanism by resorting to a simple database that maps procedure names to load modules.

5.7 Summary

Unicon's system facilities provide a high-level interface to the common features of modern operating systems, such as directories and network connections. This interface is vital to
most programs, and it also presents the main portability challenges, since a Unicon design goal is for applications to require no source code changes and no conditional code needed to run on most operating systems. Of course some application domains such as system administration are inevitably platform dependent.

There are two major areas of the system interface that are whole application domains extensive enough to warrant an entire chapter of their own: databases and graphics. Databases can be viewed as a hybrid of the file and directory system interface with some of the data structures described in Chapter 2. Since many databases are implemented using a client/server architecture, the database interface also includes aspects of networking. Databases are presented in the Chapter 6.

Graphics is another crucial component of the system interface, rich enough to warrant special features built-in to the language, and deep enough to warrant an entire book. Like databases, graphics can be viewed as an extension of the file data type described in this chapter. Unicon's powerful 2D and 3D graphics facilities are discussed in Chapter 7.
Chapter 6

Databases

Databases are technically part of the system interface described in the last chapter, but they are an important application area in their own right. Different kinds of databases are appropriate in different situations depending on how much information is to be stored and what kinds of accesses to the information are supported. This chapter describes three kinds of databases for which Unicon provides direct support, enabling you to:

- Read and write memory-based structures to data files.
- Use DBM databases as a persistent table data type.
- Manipulate SQL databases through the ODBC connection mechanism.

6.1 Language Support for Databases

Unicon provides transparent access to databases stored in local files and on remote servers. The term *transparent* means that the built-in functions and operators used to access information in a database are the same as those used to access information in the memory-based structures presented in Chapter 2. To do this, connections to databases are represented by new built-in types that are extensions of the file and table data types.

Some people might prefer a different syntax for databases from what is used for data structures. A different syntax, such as one based purely on function calls, would be consonant with the difference in performance the programmer can expect to see when accessing data in files as opposed to memory-based data structures. However, the performance of operators already depends on the type of the operands. Consider the expression ![x]. If x is a structure, its elements are generated from memory, but if x is a file, ![x] reads and generates lines from the file. The goal for Unicon is to make databases just as easy to learn and use as the rest of the language, and to minimize the introduction of new concepts.

The word “database” means different things to different people; for some, it is the short form of “relational database.” This chapter uses the term database to refer to any method
of providing persistent structures that store information from one program run to the next. The operators used to access a database determine whether one element at a time is read or written, or whether many operations are buffered and sent to the database together.

### 6.2 Memory-based Databases

If the entire database fits in memory at once, you can achieve vast speedups by avoiding the disk as much as possible. For example, all queries to read the database can be performed from memory. The database may be modified in memory immediately, and updated on the disk later on. Memory-based databases are increasingly feasible as workstation main memories grow larger, they are an excellent choice for many applications.

One way to implement a memory-based database is to build up your arbitrary structure in memory, and then use the Icon Program Library module `xcodes` to write them out and read them in. The `xcodes` procedures convert structures to a string format that can be written to a file, and convert such strings back into the corresponding structure. The following sequence saves the contents of structure `db` to a file named `db.dat`.

```icon
db := table()
db["Ralph"] := "800-USE-ICON"
db["Ray"] := "800-4UN-ICON"
dbf := open("db.dat","w")
xencode(db, dbf)
close(dbf)
```

The converse operation, reading in a structure from a file is also simple:

```icon
dbf := open("db.dat")
db := xdecode(dbf)
close(dbf)
write(db["Ralph"])
```

This approach works great for databases that do not need to be written to disk on an ongoing basis and for which the queries can readily be expressed as operations on structure types. For example, a telephone rolodex application would be well-served by this type of database. The data fits comfortably in memory, updates often occur in batches, and simple queries (such as a student’s name) provide sufficient access to the data. The other two kinds of databases in this chapter use traditional database technologies to efficiently address situations where this type of database is inadequate.

### 6.3 DBM Databases

A classic database solution on the UNIX platform is provided by the DBM family of library functions. DBM stands for Data Base Manager, and the functions maintain an association
between keys and values on disk, which is very similar to the table data type. DBM
was eventually followed by compatible superset libraries called NDBM (New Data Base
Manager) and GDBM (GNU Data Base Manager). Unicon uses GDBM on all platforms.

DBM databases are accessed via the **open()** function using mode "d" to open a database
for reading and writing, or mode "dr" for read-only access. Once opened, DBM databases
resemble the table data type and are manipulated using table operations. For example, if
d is a DBM file, **d[s]** performs a database insert/update or lookup, depending on whether
the expression is assigned a new value, or dereferenced for its current value. Values can
also be inserted into the database with **insert(d, k, v)** and read from it with **fetch(d, k)**.
**delete(d, k)** similarly deletes key k from the database. DBM databases are closed using
the **close()** function. The following example program takes a database and a key on the
command line, and writes out the value corresponding to that key.

```plaintext
procedure main(args)
    d := open(args[1], "d") | stop("can’t open ", args[1])
    write(d[args[2]])
end
```

If you are wondering why the call to **open()** isn’t followed by a call to **close()**, you
are right, it is proper to close files explicitly, although the system closes all files when the
program terminates. How would you generalize this program to accept a third command-
line argument, and insert the third argument (if it is present) into the database with the
key given by the second argument? You might easily wind up with something like this:

```plaintext
procedure main(args)
    d := open(args[1], "d") | stop("can’t open ", args[1])
    d[args[2]] := args[3]
    write(d[args[2]])
    close(d)
end
```

DBM databases are good for managing data sets with a simple organization, when the
size of the database requires that you update the database a record at a time, instead of
writing the entire data set. For example, if you wrote a Web browser, you might use a
DBM database to store the user’s set of bookmarks to Web pages of interest.

There is one basic limitation of DBM databases when compared with the table data
type that you should know about. DBM databases are string-based. The keys and values
you put in a DBM database get converted and written out as strings. This makes the
semantics of DBM databases slightly different from tables. For example, a table can have
two separate keys for the integer 1 and the string "1", but a DBM database will treat both
keys as the string "1". This limitation on DBM databases also means that you cannot use
structure types such as lists as keys or values in the database. If the type is not convertible
to string, it won’t work. You can use the functions \texttt{xencode()} and \texttt{xdecode()}, described in the previous section, to manually convert between strings and structures for storage in a DBM database if you really need this capability.

\section*{6.4 SQL Databases}

DBM is great for data that is organized around a single key, but it is inadequate for complex databases. The industry standard choice for enterprise-level data organization is the Structured Query Language (SQL). SQL is supported by every major database vendor.

Unlike DBM, a SQL database can contain multiple tables, and those tables are accessed by walking through a set of results to a query, rather than by accessing individual elements directly. SQL is designed for industrial-strength relational databases.

\subsection*{The SQL language}

The SQL language was invented by IBM and based on relational database theory developed by E.F. Codd. A database is a collection of \textit{tables}, and each table is a collection of \textit{rows}. The rows in a table contain information of various types in a set of named \textit{columns}. Rows and columns are similar to records and fields, except that they are logical structures and do not describe physical form or layout of the data. There is an ANSI standard definition of SQL, but many vendors provide extensions, and most vendors are also missing features from the ANSI standard. Unicon allows you to send any string you want to the SQL server, so you can write portable “vanilla SQL” or you can write vendor-specific SQL as needed.

SQL was originally intended for text-based interactive sessions between humans and their databases. Nowadays, SQL is primarily used “under the covers” by database applications that accommodate novice users with a graphical interface that does not require any knowledge of SQL, while supporting a SQL “escape hatch” for advanced users who may wish to do custom queries. This duality is paralleled in the Unicon language by the fact that Unicon’s built-in database operators and functions duplicate a subset of the capabilities of SQL. There are two ways to do things: using Unicon operations or using SQL statements.

SQL statements can be divided into several categories, the most prominent of which are data definition and data manipulation. When using SQL within a Unicon program, you build up string values containing SQL statements. In the following examples, the SQL is given unadorned by double quotes or other Unicon artifacts.

New tables are created with a \texttt{CREATE TABLE} statement, such as

\begin{verbatim}
create table addresses (name varchar(40), address varchar(40),
                     phone varchar(15))
\end{verbatim}

Tables have a primary key that must be unique among rows in the table. By default the primary key is the first one listed, so \texttt{name} is the primary key in table \texttt{addresses} above.
SQL’s data manipulation operations include **SELECT**, **INSERT**, **UPDATE**, and **DELETE**. **SELECT** determines the data set being operated on, picking rows and columns that form some projection of the original table. **SELECT** also allows you to combine information from multiple tables using relational algebra operations. Most databases are long-lived and evolve to include more columns of information over time. SQL’s ability to select and operate on projections is an important feature, since code that works with a certain set of columns continues to work after the database is modified to include additional columns.

**INSERT**, **UPDATE**, and **DELETE** all modify the table’s contents. **INSERT** adds new rows to a table. For example:

```sql
insert into addresses (name, address, phone) values ('Nick K', '1 Evil Empire', '(123)456-7890')
insert into addresses (name, address, phone) values ('Vic T', '23 Frozen Glade', '(900)888-8888')
```

**UPDATE** and **DELETE** can modify or remove sets of rows that match a particular criterion, as in

```sql
update addresses set address = '666 RTH, Issaquah' where name = 'Nick K'
delete from addresses where name = 'Vic T'
```

This section presented only a few aspects of the SQL language. For simple database tasks you can in fact ignore SQL and use the Unicon facilities described in the rest of this chapter. However, for more complex operations the best solution is to formulate some SQL commands to solve the problem. A full discussion of SQL is beyond the scope of this book. For more information on this topic you might want to read one of the following books: Ramez Elmasri and Shamkant Navanthe’s *Fundamentals of Database Systems*, C.J. Date and Hugh Darwen’s *A Guide to the SQL Standard*.

### Database architectures and ODBC

SQL databases are accessed through an underlying Open DataBase Connectivity (ODBC) transport mechanism. This mechanism allows the programmer to ignore the underlying architecture. Hiding this complexity from application programmers is important. The database architecture may vary from a single process accessing a local database, to client/server processes, to three or more tiers spanning multiple networks. Figure 6-1 illustrates the role played by ODBC in providing Unicon with database access in one- and two-tier configurations. While this chapter cannot present a complete guide to the construction of database systems, it provides concrete examples of writing Unicon client programs that access typical database servers.
To use Unicon’s SQL facilities, you must have several software components in place. First, you need a SQL server that supports ODBC. You can buy a commercial SQL server, or use a free server such as MySQL (www.mysql.com) or PostgreSQL (www.postgresql.org).

Second, you need an account, password, and permissions on the SQL server in order to connect to it. The details are server-dependent and outside the scope of this book.

Third, your client machine needs an ODBC driver manager, and an ODBC driver for your SQL server; these must be configured properly. The driver manager is a component that connects applications to various databases; ODBC drivers are dynamic link libraries that database vendors supply to talk to their database. Figure 6-2 shows database configuration for MySQL via the MyODBC GUI dialog box on Windows (left), and in a ~/odbc.ini file on Linux (right). In both cases, configuration involves knowing the internet server name or IP address, the port, and the database to connect to. For that triplet, you get to define a name called a DSN or data source name, which is the name that Unicon will pass in to open(). In the Windows dialog, this name is a textfield explicitly named as a DSN, while in the Linux odbc.ini file, it is at the top, inside the square brackets.

In both cases, there are a lot of additional options which are beyond the scope of this book. On Windows, each ODBC driver may have its own custom dialogs for configuration, while on Linux the odbc.ini file is more the property of the driver manager and is used to configure all the various drivers. As a fair warning, the details required in the dialogs or the exact syntax of the odbc.ini and its required entries for a given driver change slightly from time to time and are beyond the scope of this book. Consult current ODBC documentation for the driver manager on your platform and the specific database to which you are connecting.
Once you have the ODBC software set up, writing the Unicon client program to connect to your database is straightforward.

**Opening a SQL database**

To connect to a SQL database, call `open()` with mode "o". This establishes a session with a data source. The filename argument to `open()` is the data source to which you are connecting; it is associated with a particular ODBC driver, remote database server machine or IP number, and port within the ODBC driver manager. Mode "o" is followed by additional string arguments to `open()`. The first is an optional default table name used in the various functions that take a table name. Generally the arguments must then include the user name and password to use in connecting to the specified data source. Here is an example that establishes a connection to a database:

```unicon
f := open("unicondb", "o", "scott", "tiger")
```

The `open()` function returns a value that supports many of the operations of both a file and a table, or fails if the connection cannot be established. The underlying session information is shared by multiple calls to `open()` the same database. In addition to the network connection and SQL session information that is retained, each database file value maintains a current selection consisting of a set of rows corresponding to the current query,
and a cursor position within that selection. When a database is first opened, the selection consists of a null set containing no rows and no columns.

**Querying and modifying a SQL database**

Subsequent queries to the database can be made by calling `sql(db, sqlcmd)`. The `sql()` function sets the current selection within the database and places the cursor at the beginning of the set of selected rows. For example, to obtain Vic T’s phone number you might say

```sql
sql(f, "select phone from addresses where name='Vic T'")
```

Vic’s phone number is included if you use the original `select *` query, but the more specific your query, the less time and network bandwidth is wasted sending data that your client application must filter out. You should do as much work on the server (in SQL) as possible to make the client more efficient.

Since the function `sql()` transmits arbitrary SQL statements to the server, it can be used for many operations besides changing the current selection. The `sql()` function returns a null value when there is no return value from the operation, such as a `create table` statement. Its return value can be of varying type for other kinds of queries, and it can fail, for example if the SQL string is malformed or requests an impossible operation.

**Traversing the selected rows**

To walk through the rows of your current database selection, you call `fetch(db)`. The value returned is a row that has many of the operations of a record or a table, namely field access and subscripting operators. For example, if `fetch(db)` returns a row containing columns Name, Address, and Phone, you can write

```plaintext
row := fetch(db)
write(row.Name)
write(row["Address"])
```

Called with one argument, `fetch(db)` advances the cursor one position. With two arguments, `fetch(db, col)` produces a column from the current row without advancing the cursor.

**A SQL example application**

A human resources database might include two tables. One table might maintain employee information, such as names, identification numbers, and phone numbers, while another table maintains entries about specific jobs held, including employee’s ID, the pay rate, a code indicating whether pay is hourly or salaried, and the job title. Note that the SQL is embedded within a string literal.
sql(db, "create table employees (id varchar(11), name varchar(40),
    phone varchar(15))")
sql(db, "create table jobs (id varchar(11), payrate integer, is_salaried char,
    title varchar(40))")

Inserting rows into the database looks like

    sql("insert into employees (id, name, phone) values(32, 'Ray', '274-2977')")

Now, how can you print out the job title for any particular employee? If you have the
employee’s identification number, the task is easy, but let’s say you just have their name.
These are the kinds of jobs for which SQL was created. Information from the employees
table is effortlessly cross-referenced with the jobs table by the following SQL. The string
is long so it is split into two lines. A Unicon string literal spans multiple lines when the
closing doublequote has not been found and the line ends with an underscore character.

    sql(db, "select name,title from employees,jobs 
        where name='Ray' and employees.id = jobs.id")
    while write(fetch(db).Title)

SQL types and Unicon types

SQL has many data types, most of which correspond to Unicon types. CHAR and VARCHAR
 correspond to Icon strings; INTEGER and SMALLINT correspond to integers;
FLOAT and REAL correspond to reals, and so on. The philosophy is to convert between
Icon and SQL seamlessly and with minimal changes to the data format, but you should be
aware that these are not exact matches. For example, it is possible to define a FLOAT with
more precision than an Icon real, and it is easy to produce an Icon string that is longer
than the maximum allowed VARCHAR size on most SQL servers. Unicon programmers
writing SQL clients must be aware of the limitations of the SQL implementations they use.

Unicon has structure types for which there is no SQL equivalent. Values of these types
cannot be inserted into a SQL database unless you explicitly convert them to a SQL-
compatible type (usually a string) using a function such as xencode(). SQL also has types
not found in Unicon such as bit strings, timestamps, and BLOBS; they are represented by
strings, and strings are used to insert such values into SQL databases. Strings are also used
to represent out-of-range values when reading SQL columns into Unicon.

More SQL database functions

SQL databases are feature-rich enough to warrant a suite of functions in addition to those
they share with other kinds of files and databases. These functions are described in detail
in Appendix A, but some of them deserve special mention. The function dbtables(db)
is useful to obtain a listing of the data sources available within a particular database. Function \texttt{dbcolumns(db)} provides detailed information about the current table that is useful in writing general tools for viewing or modifying arbitrary SQL databases.

The functions \texttt{dbproduct(db)} and \texttt{dbdriver(db)} produce information about the DBMS on which \texttt{db} resides, and the ODBC driver software used in the connection. The function \texttt{dblimits(db)} produces the upper bounds for many DBMS system parameters, such as the maximum number of columns allowed in a table. These functions return their results as a record or list of records whose field names and descriptions are given in Appendix A.

### 6.5 Tips and Tricks for SQL Database Applications

In addition to the complexity of learning SQL itself, SQL database applications have a characteristic flavor which may or may not seem natural to the Unicon programmer.

**Operating on large files**

 Asking for 200MB of data in a remote SQL database is a good way to bring a computer to its knees. Some SQL operations are slow due to an inefficient query on the remote server, while others are slow because large amounts of data are transmitted over a limited network connection. For a fixed amount of data, operation time will vary radically depending on how it is organized; fewer, larger tuples are transmitted faster than many smaller tuples.

**Use multiple connections to nest queries**

It is common to use more than one table at once. Some times this is using SQL’s \texttt{JOIN} operation, but sometimes it is not. If you try to nest a second query inside a first one, you will quickly find that on a given connection, only one \texttt{SELECT} and one row set is maintained at a time. The second \texttt{SELECT} replaces the first, so for example:

```unicon
db := open("mydsn", "o", ...)
sql(db, "SELECT ...")
while r := fetch(db) do {
    sql(db, "SELECT ...")
    while r2 := fetch(db) do write(r2.foo)
}
does not work. Within your operating system and database server’s limits, the easy solution is to open multiple connections to your database:

```unicon
db1 := open("mydsn", "o", ...)
db2 := open("mydsn", "o", ...)
sql(db, "SELECT ...")```
6.5. TIPS AND TRICKS FOR SQL DATABASE APPLICATIONS

```unicon
while r := fetch(db) do {
    sql(db2, "SELECT ...")
    while r2 := fetch(db2) do write(r2.foo)
}
```

Dynamic records

Rows are represented as a special kind of Unicon record whose fields are determined at run-time from the names of selected columns. Record types introduced at runtime are called dynamic records, and they are useful in other contexts besides databases.

The function `constructor(rname, field, field, ...)` produces a procedure that constructs records named `rname` with the given fields. The field names can be arbitrary strings, but only legal identifiers will be subsequently accessible via the field operator `. `. 

The db library

The declaration `link db` provides simplified SQL access routines for non-SQL programmers. This library will not allow you to avoid learning SQL for long, but may ease the conversion from Unicon structure values into SQL strings for transmission over the network. The most useful of these procedures is `dbupdate()`, which sends a record (tuple) to the database. The following example updates two columns within a row returned by `fetch()`.

```unicon
row := fetch(db)
row.Name := "Bill Snyder"
row["Address"] := "6900 Tropicana Blvd"
dsupdate(db, row)
```

Of course, before a fetch can be performed, a row set must have been selected. The procedure `dbselect(db, columns, filter, order)` selects tuples containing columns from the database, with optional filter(s) and ordering rule(s). Inserting and deleting rows is performed by procedures `dbinsert()` and `dbdelete()`. The `dbinsert()` function takes two parameters for each column being inserted, the column name and then the value.

Unwritable tuples

Many SQL selections are read-only. The relational combination of columns from different tables is powerful, but the resulting selections are non-updatable. Another example of a read-only query is a `GROUP BY` query, which is usually applied before an aggregate count. Executing a `SELECT *` on a single table is updatable, but if you do something fancier, you will have to know the semantics of SQL to tell whether the result may be modified.
6.6 Summary

Databases are a standard form of persistent storage for modern applications. The notation for manipulating a database looks like a sequence of table and record operations, comprising a combination of Unicon and SQL statements. Database facilities give programmers direct access and control over the information flow to and from permanent storage.
Chapter 7

Graphics

Unicon provides a rich high level interface to 2D and 3D raster graphics, text fonts, colors, and mouse input facilities provided by an underlying system, such as the X Window System. Unicon’s graphics are portable across multiple platforms. The most important characteristics of the graphics facilities are:

- Simplicity, ease of learning
- Windows are integrated with Unicon’s existing I/O functions
- Straightforward input event model

This chapter presents Unicon’s 2D and 3D graphics facilities. Some material on 2D graphics comes from University of Arizona CS TR93-9. The 3D graphics sections come from Unicon TR 9, whose original author is Naomi Martinez. The definitive reference for the 2D graphics facilities is “Graphics Programming in Icon” by Griswold, Jeffery, and Townsend, and this book is of value for writing 3D programs. Online references for the graphics facilities also come with the software distributions.

Because different platforms have radically different capabilities, there is a trade-off between simplicity, portability, and full access to the underlying machine. Unicon aims for simplicity and ease of programming rather than full low-level access.

7.1 2D Graphics Basics

Unicon’s 2D facilities provide access to graphics displays without enforcing a particular user interface look-and-feel. Events other than keystrokes and mouse events are handled automatically by the runtime system. Chapter 17 describes the standard class library and user interface builder for Unicon applications.

Graphic interfaces are event driven; an event reading loop is the control mechanism driving the application. For example, if an application must be ready to redraw the contents of its window at all times, it may not compute for long periods without checking...
for window events. This event driven paradigm used in the underlying implementation is optional at the Unicon application level. Since Unicon windows handle many events automatically and “take care of themselves”, applications follow the event driven paradigm only when it is needed. Unicon’s extensive use of default values make simple graphics applications extremely easy to program, while providing flexibility where it is needed in more sophisticated applications.

A window is a special file opened with mode "g", appearing on-screen as a rectangular space for text and/or graphics. Windows support text I/O, much as one uses a text terminal. A simple Unicon graphics program might look like this:

```
procedure main()
    w := open("hello", "g")
    write(w, "hello, world")
    # do processing ... use w as if it were a terminal
    close(w)
end
```

Windows are open for both reading and writing, and support the usual file operations with the exceptions of `seek()` and `where()`. Unlike regular files, the `type()` of a window is "window". Like other files, windows close automatically when the program terminates, so the call to `close()` in the above example is optional.

Bit-mapped, or raster, graphics constitute a second programming model for windows. There are no programming “modes” and code that uses graphics may be freely intermixed with code that performs text operations. There are many graphics functions and library procedures, detailed in Appendices A and B.

**&window: the Default Window Argument** The keyword `&window` is a default window for graphics. `&window` starts with a null value; only window values (and `&null`) may be assigned to `&window`. `&window` is a default argument to most graphics functions and is used implicitly by various operations. If a program uses `&window`, the argument can be omitted from calls to functions such as `EraseArea()` and `WAttrib()`. The window argument is required for calls to file functions such as `write()` and `writes()` since these functions default to `&output`, not &window. The default window shortens the code for graphics-oriented programs and makes it faster.

**2D Graphics Coordinates** The 2D graphics functions use an integer coordinate system based on pixels (picture elements). Like the text coordinate system, 2D graphics coordinates start in the upper-left corner of the screen. From that corner the positive x direction lies to the right and the positive y direction moves down. Unlike text coordinates, the graphics coordinate axes are zero-based, which is to say that the very top leftmost pixel is (0,0) by default.
Angles are real numbers given in radians, clockwise starting at the 3 o'clock position. Many functions operate on rectangular regions specified by x, y, width, and height components. Width and height may be negative to extend the rectangle left or up from x and y. Screen output may be limited to a rectangle within the window called the clipping region. The clipping region is set or unset using the function `Clip()`. 

**Window Attributes** A window’s state has many *attributes* with associated values. Some values are defined by the system, while others are under program control, with reasonable defaults. When opening a window, `open()` allows string arguments after the filename and mode that specify initial values of attributes when the window is created. For example, to say hello in italics on a blue background one may write:

```plaintext
procedure main()
    w := open("hello", "g", "font=sans,italics", "bg=blue")
    write(w, "hello, world")
    # processing ...
end
```

After a window is created, its attributes are read and set using the function `WAttrib(w,s1,s2,...)`. Arguments to `WAttrib()` that have an equals sign are assignments that set the given value if possible; `WAttrib()` fails otherwise. `open()` only allows such attribute assignments. Some attributes can only be read by `WAttrib()` and not set.

String arguments to `WAttrib()` that have an attribute name but no value are queries which return the attribute value. `WAttrib()` generates a string result for each argument; a query on a single argument produces just the value of that attribute; for multiple arguments and in the case of assignment, the result is the `attr=val` form attribute assignments take. Attributes are also frequently set implicitly by the user’s manipulation of the window; for instance, cursor or mouse location or window size.

Table 7-1 lists attributes that are maintained on a per-window basis. Attribute values are string encodings. Usage refers to whether the attribute may be read, written or both. RWO and WO attributes can be assigned only when the window is opened. Although all attribute values are encoded as strings, they represent a range of window system features. The attribute `pointer` refers to mouse pointer shapes that may be changed by the application during different operations. The attribute `pos` refers to the position of the upper-left corner of the window on the screen. Screen position is specified by a string containing x,y coordinates, e.g. "pos=200,200".
Table 7-1
Canvas Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type / Example</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>pixel pair</td>
<td>Size of window</td>
<td>RW</td>
</tr>
<tr>
<td>pos</td>
<td>pixel pair</td>
<td>Position of window on screen</td>
<td>RW</td>
</tr>
<tr>
<td>canvas</td>
<td>normal, hidden</td>
<td>Canvas state</td>
<td>RW</td>
</tr>
<tr>
<td>windowlabel</td>
<td>string</td>
<td>Window label (title)</td>
<td>RW</td>
</tr>
<tr>
<td>inputmask</td>
<td>string</td>
<td>select categories of input events</td>
<td>RW</td>
</tr>
<tr>
<td>pointer</td>
<td>arrow, clock</td>
<td>Pointer (mouse) shape</td>
<td>RW</td>
</tr>
<tr>
<td>pointex,</td>
<td>pixel</td>
<td>Pointer (mouse) location</td>
<td>RW</td>
</tr>
<tr>
<td>pointery</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>display</td>
<td>device / &quot;my.cs.esu.edu:0&quot;</td>
<td>(X11) device window resides on</td>
<td>RWO</td>
</tr>
<tr>
<td>depth</td>
<td># of bits</td>
<td>Display depth</td>
<td>R</td>
</tr>
<tr>
<td>displaywidth,</td>
<td>pixel</td>
<td>Display size</td>
<td>R</td>
</tr>
<tr>
<td>displayheight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>image</td>
<td>string</td>
<td>Initial window contents</td>
<td>WO</td>
</tr>
</tbody>
</table>

7.2 Graphics Contexts

Some attributes are associated with the window itself, while others are associated with the graphics context, a set of resources used by operations that write to windows. This distinction is unimportant in simple applications but is useful in more sophisticated applications that use multiple windows or draw many kinds of things in windows. A graphics context has colors, patterns, line styles, and text fonts and sizes.

Although they are called graphics contexts, text operations use these attributes. Text is written using the foreground and background colors and font defined in the graphics context. Table 7-2 lists the attributes associated with a graphics context.

Table 7-2
Context Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type / Example</th>
<th>Description : Default</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>fg</td>
<td>color / &quot;red&quot;</td>
<td>Foreground color : black</td>
<td>RW</td>
</tr>
<tr>
<td>bg</td>
<td>color / &quot;0,0,0&quot;</td>
<td>Background color : white</td>
<td>RW</td>
</tr>
<tr>
<td>font</td>
<td>font name</td>
<td>Text font : fixed</td>
<td>RW</td>
</tr>
<tr>
<td>fheight, fwidth</td>
<td>integer</td>
<td>Text font max char height and width</td>
<td>R</td>
</tr>
<tr>
<td>leading</td>
<td>integer</td>
<td>Vertical # pixels between text lines</td>
<td>RW</td>
</tr>
<tr>
<td>ascent, descent</td>
<td>integer</td>
<td>Font height above/below baseline</td>
<td>R</td>
</tr>
<tr>
<td>drawop</td>
<td>logical op / reverse</td>
<td>Drawing operation: copy</td>
<td>RW</td>
</tr>
<tr>
<td>fillstyle</td>
<td>stippled, opaquestipped</td>
<td>Graphic fill style : solid</td>
<td>RW</td>
</tr>
</tbody>
</table>
7.2. GRAPHICS CONTEXTS

<table>
<thead>
<tr>
<th>Name</th>
<th>Type / Example</th>
<th>Description : Default</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>&quot;4,#5A5A&quot;</td>
<td>Fill pattern</td>
<td>RW</td>
</tr>
<tr>
<td>linestyle</td>
<td>onoff, doubledash</td>
<td>Drawing line style : solid</td>
<td>RW</td>
</tr>
<tr>
<td>linewidth</td>
<td>integer</td>
<td>Drawing line width</td>
<td>RW</td>
</tr>
<tr>
<td>clipx, clipy, clipw, cliph</td>
<td>integer</td>
<td>Clip rectangle position and extent: 0</td>
<td>RW</td>
</tr>
<tr>
<td>dx, dy</td>
<td>integer</td>
<td>Output coordinate translation : 0</td>
<td>R</td>
</tr>
<tr>
<td>image</td>
<td>string / &quot;flag.xpm&quot;</td>
<td>Initial window contents</td>
<td>WO</td>
</tr>
</tbody>
</table>

Binding Windows and Graphics Contexts Together Graphics contexts can be shared among windows, and multiple graphics contexts can be used on the same window. An Unicon window value is a binding of a canvas (an area that may be drawn upon) and a graphics context. A call `open(s,"g")` creates both a canvas, and a context, and binds them together, producing the binding as its return value.

`Clone(w)` creates a binding of the canvas and attributes of `w` with a new graphics context that is manipulated independently. `Clone()` also accepts any number of string attributes to apply to the new window value, as in `open()` and `WAttrib()`. After calling `Clone()`, two or more Unicon window values write to the same canvas. The cursor location is stored in the canvas, not the graphics context. Writing to the windows produces concatenated (rather than overlapping) output. Closing one of the window values removes the canvas from the screen but does not destroy its contents; the remaining binding references an invisible pixmap. The canvas is destroyed after the last binding associated with it closes.

Use of `Clone()` can significantly enhance performance for applications that require frequent graphics context manipulations.

Subwindows The function `Clone()` can also be used to create subwindows, which are canvases that reside within other windows. `Clone(w, "g", ...)` opens a 2D subwindow within `w`, and `Clone(w, "gl", ...)` opens a 3D subwindow within `w`. Applications must supply position and size attributes when they create a subwindow. Input events to a subwindow are not seen on the enclosing parent window and vice versa; both windows must be polled or supplied to `WActive()` or `select()` in order to handle input.

Coordinate Translation In 2D, context attributes `dx` and `dy` perform output coordinate translation. `dx` and `dy` take integer values and default to zero. These integers are added into the coordinates of all output operations that use the context; input coordinates in `&x` and `&y` are not translated.
7.3 Events

User input such as keystrokes and mouse clicks are called events. Many events are handled by Unicon automatically, including window redrawing and resizing, etc. Other events are put on a queue in the order they occur, for processing by the Unicon program. When reading from a window using file input functions such as `reads(w, 1)`, only keyboard events are produced; mouse and other events are dropped.

The primary input function for windows is `Event(w)`, which produces the next event for window `w` and removes it from the queue. If the event queue is empty, `Event()` waits for an event. Keyboard events are returned as strings, while mouse events are returned as integers. Special keys, such as function and arrow keys, are also returned as integers, described below. `Event()` also removes the next two elements and assigns the keywords `&x` and `&y` the pixel coordinates of the mouse at the time of the event. The values of `&x`, `&y` remain available until a subsequent call to `Event()` again assigns to them. `Event()` sets the keyword `&interval` to the number of milliseconds that have elapsed since the last event (or to zero for the first event). Keywords `&control`, `&shift`, and `&meta` are set by `Event()` to return the null value if those modifier keys were pressed at the time of the event; otherwise they fail. For resize events, `&interval` is set to zero and modifier keywords fail. Keywords associated with event processing on windows are summarized in Table 7-3:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;x</td>
<td>Mouse location, horizontal</td>
</tr>
<tr>
<td>&amp;y</td>
<td>Mouse location, vertical</td>
</tr>
<tr>
<td>&amp;row</td>
<td>Mouse location, text row</td>
</tr>
<tr>
<td>&amp;col</td>
<td>Mouse location, text column</td>
</tr>
<tr>
<td>&amp;interval</td>
<td>Time since last event, milliseconds</td>
</tr>
<tr>
<td>&amp;control</td>
<td>Succeeds of Control key pressed</td>
</tr>
<tr>
<td>&amp;shift</td>
<td>Succeeds if Shift key pressed</td>
</tr>
<tr>
<td>&amp;meta</td>
<td>Succeeds if Alt (meta) key pressed</td>
</tr>
</tbody>
</table>

**Keyboard Events** The regular keys that Unicon returns as one-letter strings correspond approximately to the lower 128 characters of the ASCII character set. These characters include the control keys, the escape key, and the delete key. Modern keyboards have many additional keys, such as function keys, arrow keys, "page down", etc. Unicon produces integer events for these special keys. A collection of symbol definitions for special keys is available in the library include file `keysyms.icn`. The most common of these are `Key_Down`, `Key_Up`, `Key_Left`, `Key_Right`, `Key_Home`, `Key_End`, `Key_PgUp`, `Key_PgDn`, `Key_F1...Key_F12`, and `Key_Insert`.
Mouse Events  Mouse events are returned from `Event()` as integers indicating the type of event, the button involved, etc. Keywords allow the programmer to treat mouse events symbolically. The event keywords are:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lpress</td>
<td>Mouse press left</td>
</tr>
<tr>
<td>&amp;mpress</td>
<td>Mouse press middle</td>
</tr>
<tr>
<td>&amp;rpress</td>
<td>Mouse press right</td>
</tr>
<tr>
<td>&amp;lrelease</td>
<td>Mouse release left</td>
</tr>
<tr>
<td>&amp;mrelease</td>
<td>Mouse release middle</td>
</tr>
<tr>
<td>&amp;rrelease</td>
<td>Mouse release right</td>
</tr>
<tr>
<td>&amp;ldrag</td>
<td>Mouse drag left</td>
</tr>
<tr>
<td>&amp;mdrag</td>
<td>Mouse drag middle</td>
</tr>
<tr>
<td>&amp;rdrag</td>
<td>Mouse drag right</td>
</tr>
<tr>
<td>&amp;resize</td>
<td>Window was resized</td>
</tr>
</tbody>
</table>

The following program uses mouse events to draw a box that follows the mouse pointer around the screen when a mouse button is pressed. The attribute `drawop=reverse` allows drawing operations to serve as their own inverse; see [Griswold98] for more about the `drawop` attribute. Function `FillRectangle()` draws a filled rectangle on the window and is described in the reference section. Each time through the loop the program erases the box at its old location and redraws it at its new location; the first time through the loop there is no box to erase so the first call to `FillRectangle()` is forced to fail by means of Unicon’s `/` operator.

```unicon
procedure main()
  &window := open("hello", "g", "drawop=reverse")
  repeat if Event() === (&ldrag | &mdrag | &rdrag) then {
    # erase box at old position, then draw at new position
    FillRectangle(x, y, 10, 10)
    FillRectangle(x := &x, y := &y, 10, 10)
  }
end
```

The program can inspect the window’s state using `WAttrib()`. Between the time the mouse event occurs and the time it is produced by `Event()`, the mouse may have moved. In order to get the current mouse location, use `QueryPointer()` (see below).

When more than one button is depressed as the drag occurs, drag events are reported on the most recently pressed button. This behavior is invariant over all combinations of presses and releases of all three buttons.
Resize events are reported in the same format as mouse events. In addition to the event code, &x, &y, &col and &row are assigned integers that indicate the window’s new width and height in pixels and in text columns and rows, respectively. Resize events are produced when the window manager (usually at the behest of the user) resizes the window; no event is generated when an Unicon program resizes its window.

**Key Release, Mouse Motion, and Window Closure Events** The canvas attribute inputmask allows programs to request three kinds of additional input events on windows. These events pose enough performance or portability obstacles that they are not produced by default. An "m" in the inputmask requests mouse motion events when no mouse button is depressed; by default only drag events are reported. If the inputmask contains a "k", events will be generated when keyboard keys are released. An inputmask attribute containing a "c" requests an event when a window closure is externally triggered, as in the case when a titlebar x button is pressed.

**Event Queue Manipulation** The event queue is an Unicon list that stores events until the program processes them. When a user presses a key, clicks or drags a mouse, or resizes a window, three values are placed on the event queue: the event itself, followed by two integers containing associated event information.

Pending(w) produces the event queue for window w. If no events are pending, the list is empty. The list returned by Pending() is attached to the window. Additional events may be added to it at any time during program execution. It is an ordinary list and can be manipulated using Unicon’s list functions and operators.

When several windows are open, the program may need to wait for activity on any of the windows. Each pending list could be checked until a nonempty list is found, but such a busy-waiting solution is wasteful of CPU time. The function Active() waits for window activity, relinquishing the CPU until an event is pending on one of the open windows, and then returns a window with a pending event. A window is said to starve if its pending events are never serviced. Active() cycles through open windows on repeated calls in a way that avoids window starvation.

### 7.4 Colors and Fonts

Unicon recognizes a set of string color names based loosely on a color naming system found in [Berk82]. The color names are simple English phrases that specify hue, lightness, and saturation values of the desired color. The syntax of colors is

```
[lightness] [saturation] [hue[ish]] hue
```

where lightness is one of: pale, light, medium, dark, or deep; saturation is one of weak, moderate, strong, or vivid; and where hue is any of black, gray, grey, white, pink, violet,
brown, red, orange, yellow, green, cyan, blue, purple, or magenta. A single space or hyphen separates each word from its neighbor. When two hues are supplied (and the first hue has no ish suffix), the resulting hue is halfway in between the two named hues. When a hue with an ish suffix precedes a hue, the resulting hue is three-fourths of the way from the ish hue to the main hue. When adding ish to a word ending in e, the e is dropped (for example, purple becomes purplish); the ish form of red is reddish. Mixing radically different colors such as yellow and purple does not usually produce the expected results. The default lightness is medium and the default saturation is vivid. Note that human perception of color varies significantly, as do the actual colors produced from these names on different monitors.

Color Coordinate Systems and Gamma Correction In addition to the standard color names, platform-specific color names may be supported. Colors may also be specified by strings encoding the red, green, and blue components of the desired color. Unicon accepts the hex formats "#rgb" in which r, g, and b are 1 to 4 hex digits each. Red, green, and blue may also be given in decimal format, separated by commas. The components are given on a linear scale from 0 to 65535, although displays typically offer far less precision and nonlinear colors. For example, "bg=32767,0,0" requests a medium red background; if the display is incapable of such, it approximates it as closely as possible from the available colors. "fg=0,65000,0" requests a vivid green foreground. If colors appear darker than they should, the window system is not providing linear colors. Unicon can be told to perform the correction by means of the gamma attribute; 1.0 is a default (no gamma correction), and experimenting with values between 2 and 3 usually provides satisfactory results.

Fonts

Fonts are specified by a comma-separated string of up to four fields supplying the font’s family name, followed by optional size or italic or bold designations in any order. The fonts available vary widely from system to system. Four font family names available on all Unicon systems include serif, sans, typewriter, and mono. These families map onto the system’s closest approximation of Times, Helvetica, Courier, and a monospaced console font. Font sizes are given in pixel height.

7.5 Images, Palettes, and Patterns

DrawImage(x, y, s) draws a rectangular area using an image string. String s has the form "width,palette,data". width is the width of the rectangle drawn, palette is a code that defines the colors corresponding to data values, and the rest of the data supplies pixel values. Predefined palettes and palette functions help to provide this capability. Image and palette functions are described fully in [Griswold98].
The context attribute \texttt{fillstyle} determines the pixels used by draw and fill functions like \texttt{FillPolygon()}. If the \texttt{fillstyle} is not \texttt{solid}, a pattern in the filled area is drawn in the foreground color; other pixels are drawn in the background color ("\texttt{fillstyle=textured}"). The function \texttt{Pattern(w,s)} associates a pattern denoted by \texttt{s} with \texttt{w}'s context. String \texttt{s} is a built-in pattern name, or a representation of bits that define the pattern. Pattern representations are of the form "\texttt{width,#bits}" where 1 <= width <= 32. The window system may limit the pattern's width and height to as little as 8.

The height of the pattern is defined by the number of rows in the bits component of the pattern string. Bits consists of a series of numbers, each supplying one row of the pattern, in hexadecimal format. Each digit defines four bits and each row is defined by the number of digits required to supply width bits. For example, the call

\texttt{Pattern("4,#5A5A")}

defines a 4x4 pattern where each row is defined by one hex digit.

\textbf{pme: a pixmap editor}

A simple image editor called \texttt{pme} demonstrates event processing including mouse events. \texttt{pme} displays both a small and a "magnified" display of the image being edited, allows the user to set individual pixels, and allows the user to save the image; it is well-suited for constructing and hand-editing small images such as icons and textures for use in larger 2D or 3D scenes. \texttt{pme} consists of four procedures and employs several graphics functions. A sample screen image of \texttt{pme} is presented in Figure 7-1. The "real" image is in the upper left corner; underneath it is a mouse icon which shows what color is drawn by each of the mouse buttons.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{pme_editing.png}
\caption{\texttt{pme} editing a 32x32 image}
\end{figure}

\texttt{pme} starts by declaring and initializing several variables.

\begin{verbatim}
link dialog, file_dlg
\end{verbatim}

\begin{verbatim}
global lmargin, colors, colorbinds
\end{verbatim}

\begin{verbatim}
procedure main(argv)
    local i := 1, j, s, e, x, y, width := 32, height := 32
\end{verbatim}
7.5. IMAGES, PALETTES, AND PATTERNS

The image width and height can be specified on the command line with a **-size** option, for example, **pme** **-size** 16,64.

```plaintext
if argv[1]=="-size" then {
  i +:= 1
    width := integer(tab(many(&digits))) | stop("bad -size")
    ="," | stop("bad -size")
    height := integer(tab(0)) | stop("bad -size")
    i +:= 1
  }
}
```

Following the size arguments, **pme** checks for a filename specifying the bitmap to edit. If one is found, it is read into the regular scale image, and then the magnified scale image is constructed by reading each pixel using the function **Pixel()**, and filling an 8x8 rectangle with the corresponding color.

```plaintext
i := j := 0
every p := Pixel(0, 0, width, height) do {
  Fg(p)
  FillRectangle(j * 8 + lmargin + 5, i * 8, 8, 8)
  j +:= 1
  if j = width then { i +:= 1; j := 0 }
}
```

After the images are loaded with their initial contents, if any, a grid is drawn on the magnified image to delineate each individual pixel’s boundary. The user’s mouse actions within these boxes change the colors of corresponding pixels in the image. An list of three bindings to the window, each with an independently-set foreground color, is used to represent the color settings of the mouse buttons.

```plaintext
colors := [Clone("fg=red"),Clone("fg=green"),Clone("fg=blue")]
```

The main event processing loop of **pme** is simple: Each event is fetched with a call to **Event()** and immediately passed into a case expression. The keystroke "q" exits the program; the keystroke "s" saves the bitmap in a file by calling **WriteImage()**, asking for a file name if one has not yet been supplied.

```plaintext
case e := Event() of {
  "q"|"\e": return
  "s"|"S": {
    if /s | (e=="S") then s := getfilename()
    write("saving image ", s, "," size ", width,"," , height)
    WriteImage( s, 0, 0, width, height)
  }
}
```
Mouse events result in drawing a single pixel in both the magnified and regular scale bitmaps using one of the colors depicted on the mouse icon.

\[
\text{\&lpress | \&ldrag | \&mpress | \&mdrag | \&rpress | \&rdrag : \{ \\
  x := (\&x - \text{lmargin} - 5) / 8 \\
  y := \&y / 8 \\
  \text{if (} y < 0 \text{) | (} y > \text{height-1) | (} x > \text{width) then next } \\
  \text{if } x \geq 0 \text{ then dot}(x, y, (-e - 1) \% 3)
\]

To change the color drawn by a mouse button, you click on it.

\[
\text{else \{ # x < logical 0. User clicked in mouse area } \\
  \text{if } \&x < 21 \text{ then getacolor(1, "left") } \\
  \text{else if } \&x < 31 \text{ then getacolor(2, "middle") } \\
  \text{else getacolor(3, "right") } \\
  \text{until Event() \=== (} \&mrelease | \&lrelease| \&rrelease) \\
\}
\]

Pixel drawing is handled by procedure dot(), whose third argument specifies which button, and therefore which color to draw. The dot is drawn using FillRectangle() in the magnified window; in the regular scale window DrawPoint() suffices.

\[
\text{procedure dot}(x, y, c) \\
  \text{if } (x|y) < 0 \text{ then fail } \\
  \text{FillRectangle(colors[c+1], } x*8 + \text{lmargin} + 5, y*8, 8, 8) \\
  \text{DrawPoint(colors[c+1], } x, y) \\
  \text{DrawRectangle}(x*8 + \text{lmargin} + 5, y*8, 8, 8)
\]

pme illustrates several aspects of the Unicon graphics facilities. Note the event-handling: a case expression handles various keystrokes and mouse events with simpler control structure than in most languages’ GUI event processing.

Listing 7-1 pme: a Unicon bitmap editor

\[
\text{link dialog} \\
\text{link file_dlg} \\
\text{global lmargin, colors} \\
\text{procedure main(argv)} \\
  \text{local i := 1, j, s, e, x, y, width := 32, height := 32 } \\
  \text{if argv[1]=="-size" then \{ } \\
  \text{i += 1 } \\
  \text{argv[2] ? \{ } \\
  \text{width := integer(tab(many(&digits))) | stop("bad -size")}
\]
7.5. IMAGES, PALETTES, AND PATTERNS

="," | stop("bad -size")
height := integer(tab(0)) | stop("bad -size")
i += 1
}

lmargin := max(width, 65)
if s := argv[i] then {
  &window := open("pme", "g", "image="||s) | stop("cannot open window")
  width <=: WAttrib("width")
  height <=: WAttrib("height")
  lmargin := max(width, 65)
  WAttrib("size="||(width*8+lmargin+5)||","||(height*8))
  i := j := 0
  every p := Pixel(0, 0, width, height) do {
    Fg(p)
    FillRectangle(j * 8 + lmargin + 5, i * 8, 8, 8)
    j += 1
    if j = width then { i += 1; j := 0 }
  }

} else {
  &window := open("pme", "g", "size=" || (lmargin+width*8+5)||","||(height*8+5)) | stop("cannot open window")
}

colors := [Clone("fg=red"),Clone("fg=green"),Clone("fg=blue")]
every i := 1 to 3 do {
  DrawArc(4+i*10, height+68, 7, 22)
  FillArc(colors[i], 5+i*10, height+70, 5, 20)
}

DrawRectangle( 5, height+55, 45, 60, 25, height+50, 5, 5)
DrawCurve(27, height+50, 27, height+47, 15, height+39,
        , 40, height+20, 25, height+5)
Fg("black")
every i := 0 to height-1 do
every j := 0 to width-1 do
  DrawRectangle(j*8+lmargin+5, i*8, 8, 8)
DrawLine(0, height, width, height, width, 0)
repeat {
case e := Event() of {
  "q"|"\e": return
  "s"|"S": {
    if /s | (e=="S") then s := getfilename()
    write("saving image ", ", " size ", width,"", height)
    WriteImage( s, 0, 0, width, height)
7.6 3D Graphics

Three-dimensional graphics are provided in Unicon on platforms which support the industry standard OpenGL libraries. Unicon provides the basic primitives, transformations, lighting, and texturing elements of 3D computer graphics in a simplified fashion, providing a good basis to rapidly construct 3D scenes. The Unicon 3D interface consists of sixteen new functions and six functions that were extended from the 2D graphics facilities, compared with OpenGL’s 250+ functions. While Unicon’s 3D interface vastly simplifies some aspects of 3D programming compared with the OpenGL C interface, it does not currently provide
access to several features of OpenGL including blending, fog, anti aliasing, display lists, selection, and feedback.

This section explains in detail how to use Unicon’s 3D facilities, for programmers who already have some idea of how 3D graphics work. A 3D window is opened using mode "gl" and is very similar to a 2D window, so many ideas earlier in this chapter are needed for 3D programming. 3D graphics use the 2D windowing functions and attributes and introduce several new ones.

A primary difference between 2D and 3D is that graphics operations in 2D windows use (x,y) integer pixel coordinates relative to the upper left corner of the window, while 3D windows use (x,y,z) real number coordinates in an abstract geometric world. A mobile viewer’s position, and the direction they are looking, determine what is visible. Coordinates of 3D objects go through a series of translations, scalings, and rotations to determine their final location; these matrix transformations are used to compose aggregate objects from their parts. In addition to the coordinate system difference, 3D scenes usually employ a rich lighting model, and use materials and textures to draw objects more frequently than a solid foreground color. For this reason, the \texttt{fg} attribute is extended in the 3D realm to denote a foreground \textit{material}, including color as well as how the object appears in different types of lighting.

**Opening windows for 3D graphics**

To open a 3D graphics window, call the built in function \texttt{open()}, passing in the title of the window to be opened and mode "gl".

\begin{verbatim}
W := open("win", "gl")
\end{verbatim}

As in the 2D facilities, if a window is assigned to the keyword variable \&window, it is a default window for subsequent 3D function calls.

**3D attributes**

Features such as lighting, perspective, textures, and shading give a scene the illusion of being three-dimensional. A Unicon programmer makes use of context attributes to control these features. By assigning new values to various attributes, the programmer controls many aspects of the scene. Attributes to control the coordinate system, field of view, lighting and textures are included in the Unicon 3D graphics facilities.

Some of the most basic context attributes concern the coordinate system. In 3D graphics, x-, y-, and z-coordinates determine where to place an object. The objects that are visible on the screen depend on several things, the eye position, the eye direction, and the orientation of the scene. If these items are not taken into account, the scene desired by the user and the scene drawn may be two very different things.
To think about these attributes, imagine a person walking around a 3D coordinate system. What the person sees becomes the scene viewed on the screen. The eye position specifies where the person is standing. Things close to the person appear larger and seem closer than objects further away. The eye direction specifies where the person is looking. If the person is looking toward the negative z-axis, only the objects situated on the negative z-axis are viewed in the scene. Anything on the positive z-axis is behind the viewer. Finally, the up direction can be described by what direction is up for the person.

The eye position is given by the attribute eyepos. By default this is set to be at the origin or (0, 0, 0). The eye direction is given by the attribute eyedir. By default this is set to be looking at the negative z-axis. The up direction can be specified by the attribute eyeup and by default is (0, 1, 0). The attribute eye allows the user to specify eyepos, eyedir, and eyeup with a single value. Changing any of these attributes causes the scene to redraw itself with the new eye specifications.

Table 7.5 lists the added context attributes used on 3D windows.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type / Example</th>
<th>Description : Default</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>boolean / &quot;on&quot;</td>
<td>Buffer mode : off</td>
<td>RW</td>
</tr>
<tr>
<td>dim</td>
<td>integer / 3</td>
<td>Dimension : 2</td>
<td>RW</td>
</tr>
<tr>
<td>eye</td>
<td>xyz nonuple / &quot;0,0,0,0,0,0,0,0,0&quot;</td>
<td>Eye position,direction,up : &quot;0,0,0,0,0,-1,0,1,0&quot;</td>
<td>RW</td>
</tr>
<tr>
<td>eyedir</td>
<td>xyz triple / &quot;0,0,0&quot;</td>
<td>Eye direction/target : &quot;0,0,-1&quot;</td>
<td>RW</td>
</tr>
<tr>
<td>eyepos</td>
<td>xyz triple / &quot;0,0,0&quot;</td>
<td>Eye position : &quot;0,0,0&quot;</td>
<td>RW</td>
</tr>
<tr>
<td>eyeup</td>
<td>xyz triple / &quot;0,0,0&quot;</td>
<td>Eye up vector : &quot;0,1,0&quot;</td>
<td>RW</td>
</tr>
<tr>
<td>meshmode</td>
<td>string / &quot;triangles&quot;</td>
<td>Polygon mesh mode : &quot;polygon&quot;</td>
<td>RW</td>
</tr>
<tr>
<td>normals</td>
<td>real array</td>
<td>Normal vectors: n/a</td>
<td>RW</td>
</tr>
<tr>
<td>rings</td>
<td>integer</td>
<td>Number of rings in spheres/cylinders : 10</td>
<td>RW</td>
</tr>
<tr>
<td>selection</td>
<td>boolean / &quot;off&quot;</td>
<td>Selection</td>
<td>RW</td>
</tr>
<tr>
<td>slices</td>
<td>integer</td>
<td>Number of slices in spheres, cylinders : 15</td>
<td>RW</td>
</tr>
<tr>
<td>texcoord</td>
<td>vector of reals</td>
<td>Texture (u,v) coordinates</td>
<td>RW</td>
</tr>
<tr>
<td>texture</td>
<td>image</td>
<td>Texture</td>
<td>RW</td>
</tr>
</tbody>
</table>

### 3D drawing primitives

In 2D, programs draw points, lines, polygons, and circles. Functions that have been extended for 3D include `DrawPoint()`, `DrawLine()`, `DrawSegment()`, `DrawPolygon()`, and `FillPolygon()`. The 3D facilities introduce many new primitives, including cubes, spheres, tori, cylinders, and disks. These are described in Table 7-6 below.
Many scenes are drawn using a mixture of 2D, 3D and 4D objects. The context attribute \texttt{dim} allows the program to switch between the different dimensions when specifying the vertices of an objects. A user can draw 2D, 3D, or 4D objects by assigning \texttt{dim} the values of 2, 3, or 4. For primitives that take \(x\), \(y\), and \(z\) coordinates, specifying only \(x\) and \(y\) coordinate is not sufficient. For this reason, \texttt{"dim = 2"} disallows the use of these primitives. These functions are \texttt{DrawSphere()}, \texttt{DrawTorus()}, \texttt{DrawCube()}, and \texttt{DrawCylinder()}. By default the value of \texttt{dim} is three.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Function</th>
<th>Parameters</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube</td>
<td>\texttt{DrawCube()}</td>
<td>(x), (y), and (z) coordinates of the lower left front corner, and the length of the sides.</td>
<td><img src="image" alt="Cube" /></td>
</tr>
<tr>
<td>Cylinder</td>
<td>\texttt{DrawCylinder()}</td>
<td>(x), (y), and (z) coordinates of the center, the height, the radius of the top, the radius of the bottom. If one radius is smaller than the other, a cone is formed.</td>
<td><img src="image" alt="Cylinder" /></td>
</tr>
<tr>
<td>Disk</td>
<td>\texttt{DrawDisk()}</td>
<td>(x), (y), and (z) coordinates of center, the radius of the inner circle, and the radius of the outer circle. An additional two angle values specify a partial disk.</td>
<td><img src="image" alt="Disk" /></td>
</tr>
<tr>
<td>Solid Polygon</td>
<td>\texttt{FillPolygon()}</td>
<td>(x), (y), and (z) coordinates of each vertex of the polygon.</td>
<td><img src="image" alt="Solid Polygon" /></td>
</tr>
<tr>
<td>Line</td>
<td>\texttt{DrawLine()}</td>
<td>(x), (y), and (z) coordinates of each vertex.</td>
<td><img src="image" alt="Line" /></td>
</tr>
</tbody>
</table>
### Coordinate transformations

Matrix multiplications are used to calculate transformations such as rotations on objects and the field of view. Functions to perform several matrix operations in support of coordinate transformation are available. The main transformation functions are $\text{Translate}(dx,dy,dz)$, $\text{Scale}(mx,my,mz)$, and $\text{Rotate}(a,x,y,z)$.

In many 3D graphics applications, transformations are composed as the pieces of an object are drawn relative to one another. Transformations are saved and restored as objects are traversed. Unicon uses the system's matrix stacks to keep track of the current matrix with a stack of matrices, where the top of the stack is the current matrix. Several functions manipulate the matrix stack. The function $\text{PushMatrix()}$ pushes a copy of the current matrix onto the stack. By doing this the user can compose several different transformations. The function $\text{IdentityMatrix()}$ changes the current matrix to the identity matrix. To discard the top matrix and to return to the previous matrix, the function $\text{PopMatrix()}$ pops the top matrix off the matrix stack.

There are different matrix stacks for the projection and model view. The projection stack contains matrices that perform calculations on the field of view, based on the current eye attributes. If these eye attributes are changed, previous manipulations of the projection matrix stack are no longer valid. The maximum depth of the projection matrix stack is two. Trying to push more than two matrices onto the projection matrix stack will generate a runtime error. The model view stack contains matrices to perform calculations on objects within the scene. Transformations on the model view stack affect the subsequently drawn objects. The maximum depth of this stack is 32; pushing more than 32 matrices on the model view stack results in an error. Furthermore, only one matrix stack can be

<table>
<thead>
<tr>
<th>Shape</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygon</td>
<td><code>DrawPolygon()</code></td>
<td>x, y, and z coordinates of each vertex.</td>
</tr>
<tr>
<td>Point</td>
<td><code>DrawPoint()</code></td>
<td>x, y, and z coordinates of each point.</td>
</tr>
<tr>
<td>Segment</td>
<td><code>DrawSegment()</code></td>
<td>x, y, and z coordinates of each vertex.</td>
</tr>
<tr>
<td>Sphere</td>
<td><code>DrawSphere()</code></td>
<td>x, y, and z coordinates of center and the radius of the sphere.</td>
</tr>
<tr>
<td>Torus</td>
<td><code>DrawTorus()</code></td>
<td>x, y, and z coordinates of the center, an inner radius and an outer radius.</td>
</tr>
</tbody>
</table>
manipulated at any given time. The function `MatrixMode()` switches between the two matrix stacks.

### Lighting and materials

Lighting is important in making a graphics scene appear to be 3D. Adding lighting to a scene can be complicated and the hardware support for lighting is at present a very crude approximation. Light sources emit different types of light. *Ambient* light has been scattered so much that is difficult to determine the source; backlighting in a room is an example. *Diffuse* light comes from one direction and is central in defining what color the object appears to be. Finally, *specular* light not only comes from one direction, but also tends to bounce off the objects in the scene.

Applications control lighting using context attributes set using `WAttrib()`. For a 3D scene in Unicon, eight lights are available. Attributes `light0 - light7` control the eight lights. Each light can be turned on or off and has a position and lighting value. A lighting value is a string which contains one or more semi-colon separated lighting properties. A lighting property is of the form

```
[diffuse|ambient|specular] color name
```

A new lighting value can be specified without turning the light on or off. The following call turns `light1` on and gives it diffuse yellow and ambient gold lighting properties.

```
WAttrib(w, "light1=on, diffuse yellow; ambient gold")
```

The following expression sets `light0` to the default values for the lighting properties.

```
WAttrib(w, "light0=diffuse white; ambient black; _
specular white; position 0.0, 1.0, 0.0")
```

Interacting with the lights, the objects in a scene may have several material properties. The material properties are ambient, diffuse, and specular, which are similar to the light properties, plus emission, and shininess. If an object has an emission property, it emits light of a specific color. Using combinations of these material properties one can give an object the illusion of being made of plastic or metal.

In 2D, the foreground color is controlled using the context attribute `fg` and set with `Fg()` or `WAttrib()`. In 3D, the attribute `fg` is extended to allow a semi-colon separated list of material properties with the color that property should have. A programmer can specify a material property as a simple color value or by providing comma-separated red, green, and blue intensities as real numbers between 0.0 and 1.0. More general material properties are of the form

```
[ diffuse | ambient | specular | emission ] color name
```
or "shininess n", where n is some integer in the range 0 ≤ n ≤ 128.

The default material property type is diffuse, so the call \texttt{Fg("red")} is equivalent to \texttt{Fg("diffuse red")}. For shininess, a value of 0 spreads specular light broadly across an object and a value of 128 focuses specular light at a single point. The following line of code changes the current material property to diffuse green and ambient orange.

\begin{verbatim}
WAttrib(w, "fg=diffuse green; ambient orange")
\end{verbatim}

The default values of the material properties are given in the following example.

\begin{verbatim}
Fg(w, "diffuse light grey; ambient grey; specular black; emission black; _
shininess 50")
\end{verbatim}

Using lights and materials in Unicon was simplified by extending the design of the 2D graphics facilities. The \texttt{fg} attribute greatly reduces the number of lines of code needed for a scene. Thanks to this design along with the extensive use of defaults, a programmer can use lighting in a 3D graphics application without much effort.

### 7.7 Textures

Another important area of 3D graphics is textures. Adding textures to a scene can give a scene a realistic feel. There are several aspects to using textures. A texture is a rectangular image that is "glued" onto objects in a scene. The appearance of the textured objects in the scene depends on several pieces of information supplied by the programmer. These include the texture image and what parts of the texture image is mapped to what parts of the object.

The attribute \texttt{texmode} enables or disables textures, which are disabled by default. \texttt{WAttrib("texmode=on")} enables textures. When textures are enabled and a texture image is given, the texture is applied to the objects drawn in the scene.

Unicon provides several formats to specify a texture image. A texture can be a Unicon window, an image file, or a string. String textures are encoded in one of the language standard formats \texttt{"width,pallet,data"} or \texttt{"width,#data"} described in the 2D graphics facilities. In the first case the pallet will determine what colors appear in the texture image. In the second case, the foreground color and background color are used. The ability to use another Unicon window as a texture provides great flexibility for texture images, allowing programs to create texture images dynamically.

Textures must have a height of $2^n$ pixels and width of $2^m$ pixels where n and m are integers. If not, the texture dimensions are automatically scaled down to the closest power of 2. Rescaling affects application performance and may cause visual artifacts, so it may be wise to create textures with appropriate sizes in the first place. Examples of how to use textures specified in the different forms are given below.
A programmer specifies a texture either by calling \texttt{WAttrib("texture=...") or using \texttt{Texture(t)}. These methods differ only in that a window cannot be used as a texture with \texttt{WAttrib()}, so \texttt{Texture()} must be called when a window is used as a texture.

A program can specify how a texture is applied to a particular object by specifying texture coordinates and vertices. Texture coordinates are \(x\) and \(y\) coordinates within the texture; texture coordinate \((0.0, 0.0)\) is the lower left corner of the texture image. Texture coordinates are mapped to the vertices of an object in the scene. Together, the texture coordinates and the vertices determine what the object looks like after textures have been applied. Since texture coordinates are complex, defaults are provided. Assigning attribute \texttt{texcoord} the value \texttt{auto} causes system default texture coordinates to be used. The defaults are dependent on the type of primitive.

Non-default texture coordinates are given in several ways, such as \texttt{WAttrib("texcoord=s")}, where \(s\) is a comma separated string of real number values between 0.0 and 1.0. Each pair of values is taken as one texture coordinate; there must be an even number of real values or the assignment of texture coordinates fails. One can assign texture coordinates by calling \texttt{Texcoord(x1,y1,...)} where \(x\) and \(y\) are real number values between 0.0 and 1.0. Finally one can use \texttt{Texcoord(L)} where \(L\) is a list of real number texture coordinates. The texture coordinates given by the programmer are used differently depending on the type of primitive to be drawn. If the primitive is a point, line, line segment, polygon, or filled polygon, then a texture coordinate given is assigned to each vertex. If there are more texture coordinates than vertices, unused texture coordinates are ignored. If there are more vertices than texture coordinates the application of a texture will fail. In order to use non-default texture coordinates with cubes, tori, spheres, disks, and cylinders a programmer should approximate the desired mapping with filled polygons. These specifications are given in Table 7-7.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Default Texture Coordinates (from [OpenGL00] chapter 6)</th>
<th>Effect of Texture Coordinates</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube</td>
<td>The texture image is applied to each face of the cube.</td>
<td>None</td>
<td><img src="image" alt="Cube" /></td>
</tr>
<tr>
<td>Shape</td>
<td>Description</td>
<td>Texture Assignment</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td>Sphere</td>
<td>The y texture coordinate ranges linearly from 0.0 to 1.0. On spheres this is from $z = -\text{radius}$ to $z = \text{radius}$; on cylinders, from $z = 0$ to $z = \text{height}$. The x texture coordinate ranges from 0.0 at the positive y-axis to 0.25 at the positive x-axis, to 0.5 at the negative y-axis to 0.75 at the negative x-axis back to 1.0 at the positive y-axis.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Cylinder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filled Polygon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>The x and y texture coordinates are given by $p_1x_0 + p_2y_0 + p_3z_0 + p_4w_0$.</td>
<td>A texture coordinate is assigned to a vertex.</td>
<td></td>
</tr>
<tr>
<td>Polygon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.7. TEXTURES

| Torus | The x and y texture coordinates are given by $p_1x_0 + p_2y_0 + p_3z_0 + p_4w_0$ | None | ![Image] |

3D Examples

Changing Context Attributes The user can change attributes throughout a program. Multiple attributes can be changed with one call to `WAttrib()`. The following line of code changes the eye position to (0.0, 0.0, 5.0) and the eye direction to look at the positive z-axis. An assignment to `eyepos`, `eyedir`, `eyeup` or `eye` redraws the screen; a given call to `WAttrib()` will only redraw the scene once.

```
WAttrib("eyepos=0.0,0.0,5.0","eyedir=0.0,0.0,1.0")
```

The values of the attributes can also be read by using the function `WAttrib()`. The current eye position could be stored in variable `ep` by the call:

```
ep := WAttrib("eyepos")
```

Drawing Primitives Here is an example that uses some of the drawing primitives.

```
Fg(w, "ambient yellow")
DrawDisk(w, 0.4, -0.5, -4.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.5, -5.0, 0.5, 1.0)
Fg(w, "diffuse white")
DrawDisk(w, 0.4, -0.5, -4.0, 0.0, 1.0, 0.0, 225.0,1.0, 0.5, -5.0, 0.5,1.0,0.0,125.0)
Fg(w, "ambient pink")
DrawCylinder(w, 0.0, 1.0, -5.0, 1.0, 0.5, 0.3)
Fg(w, "specular navy")
DrawDisk(w, -0.5, -0.5, -2.0, 0.5, 0.3)
Fg(w, "emission green")
DrawSphere(w, 0.5, 1.0, -3.0, 0.5)
WAttrib(w, "light0=on, diffuse white")
```
The function $F_g()$ specifies the material properties of subsequently drawn objects that affect their color and appearance. In this example, a cube with a diffuse green material is drawn with sides of length 0.7. Then a sphere with a diffuse purple and ambient blue material is drawn with radius 0.5 and center (0.4, -0.5, -4.0). Next a diffuse yellow and ambient grey torus with center (-1.0, 0.4, -4.0), an inner radius of 0.4, and an outer radius of 0.5 is drawn. Finally a filled polygon with a diffuse red material property and three vertices, (0.25, -0.25, -1.0), (1.0, 0.25, -4.0) and (1.3, -0.4, -3.0) is drawn.

Different Types of Lighting  The next example shows the difference between the different types of lighting that can be used in a scene. Each window is the same scene rendered using different lighting. The upper right scene has an ambient blue-green light. The upper left scene was drawn using a diffuse blue-green light. The lower right scene uses only a specular blue-green light. The scene in the lower left uses all three types of lighting.

```plaintext
w := open("ambient.icn","gl","bg=black","size=400,400")
WAttrib(w,"light0=on, ambient blue-green","fg=specular white")
DrawCylinder(w, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
DrawTorus(w,0.0, -0.2, -3.5, 0.3, 0.7)
DrawSphere(w,0.0, 0.59, -2.2, 0.3)

x := open("diffuse.icn","gl","bg=black","size=400,400")
WAttrib(x,"light0=on, diffuse blue-green","fg=specular white")
DrawCylinder(x, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
DrawTorus(x,0.0, -0.2, -3.5, 0.3, 0.7)
DrawSphere(x, 0.0, 0.59, -2.2, 0.3)

y := open("specular.icn","gl","bg=black","size=400,400")
WAttrib(y,"light0=on,specular blue-green","fg=specular white")
DrawCylinder(y, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
DrawTorus(y,0.0, -0.2, -3.5, 0.3, 0.7)
DrawSphere(y, 0.0, 0.59, -2.2, 0.3)

z := open("all.icn","gl","bg=black","size=400,400")
WAttrib(z,"light0=on, diffuse blue-green; _
```
specular blue-green; ambient blue-green","fg=specular white")
DrawCylinder(z, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
DrawTorus(z, 0.0, -0.2, -3.5, 0.3, 0.7)
DrawSphere(z, 0.0, 0.59, -2.2, 0.3)

Figure 7-4 shows the effects of emission color on an object.

Figure 7-4: Mixing Emission and Diffuse Material Properties

Fg(w, "emission blue; diffuse yellow")
DrawSphere(w, -1.5, 1.0, -5.0, 0.7)
Fg(w, "emission black")
DrawSphere(w, 0.0, 0.0, -5.0, 0.7)
Fg(w, "emission red")
DrawSphere(w, 1.5, -1.0, -5.0, 0.7)

In the above example, three yellow spheres are drawn. An emission color of blue makes the sphere appear white with a blue ring. With a red emission color, the sphere remains yellow, but now has an orange-red ring. The middle sphere shows the effect of having no emission color. In order to obtain the diffuse yellow sphere in the center, the emission color was changed to black, without changing the diffuse property.

Textures This section contains examples of the use of textures in a scene. The following example uses a file as a texture. A .gif image of a map of the world is used to texture a torus using the default texture coordinates.
Instead of using `WAttrib(w, "texture=map.gif")` to specify the .gif file, a call to `Texture(w, "map.gif")` could be used to obtain the same result.

The next example uses an image string to specify a texture image. The string used for this example is taken from Graphics Programming in Icon [Griswold98] page 156. This string is used as a texture on a cube using the default texture coordinates.

```
WAttrib(w, "texmode=on")
sphere:= "16,g16, FFFFB98788AEFFFF" ||
 "FFD865554446AFFF FD856886544339FF E8579BA9643323AF"||
 "A569DECA7433215E 7569CDB86433211A 5579AA9643222108"||
 "4456776533221007 44444333210007 433333222100008"||
 "533322221100000A 822222111000003D D41111100000019F"||
 "FA200000000018EF FFA400000028EFF FFFD9532248BFFFF"
Texture(w, sphere)
DrawCube(w, 0.0, 0.0, -3.0, 1.2)
```
The next example shows the use of a Unicon window as a texture. An image of a lamp is drawn on the first window in \textit{gl} mode. This window is then used as a texture on a cylinder. The same method can be used to embed 2D window contents in 3D scenes. Note that in the following code the first window is opened with size 256 x 256. Texture images must have height and width that are powers of 2, or the system must rescale them. The default coordinates for cylinders are used.

![Figure 7-7: A Texture Obtained from Another Window’s Contents](image)

```icon
w := open("win1","gl","bg=light blue","size=256,256")
Fg(w, "emission pale grey")
PushMatrix(w)
Rotate(w, -5.0, 1.0, 0.0, 0.0)
DrawCylinder(w, 0.0, 0.575, -2.0, 0.15, 0.05, 0.17)
PopMatrix(w)
Fg(w, "diffuse grey; emission black")
PushMatrix(w)
Rotate(w, -5.0, 1.0, 0.0, 0.0)
DrawCylinder(w, 0.0, 0.0, -2.5, 0.7, 0.035, 0.035)
DrawTorus(w, 0.0, -0.22, -2.5, 0.03, 0.06)
DrawTorus(w, 0.0, 0.6, -2.5, 0.05, 0.03)

w2 := open("win2.icn","gl","bg=black","size=400,400")
WAttrib(w2, "texmode=on")
Texture(w2, w)
Fg(w2, "diffuse purple; ambient blue")
DrawCylinder(w2, 0.0, 0.0, -3.5, 1.2, 0.7, 0.7)
```

The next two examples illustrate the use of the default texture coordinates versus texture coordinates specified by the programmer. In both examples, a bi-level image is used as the texture image. The format for such a string is described in section 2.7. This image is taken from Graphics Programming in Icon page 159. The first example uses the default texture coordinates for a filled polygon, which in this case is just a square with sides of length one.
In this case the default texture coordinates are as follows. The coordinate (0.0, 0.0) of the texture image is mapped to the vertex (0.0, 0.0, -2.0) of the square, (0.0, 1.0) is mapped to (0.0, 1.0, -2.0), (1.0, 1.0) is mapped to (1.0, 1.0, -2.0), and (1.0, 0.0) is mapped to (1.0, 0.0, -2.0).

This example uses the same texture image and the same object to be textured, but instead uses the texture coordinates (0.0, 1.0), (1.0, 1.0), (1.0, 1.0), and (1.0, 0.0). So the coordinate (0.0, 1.0) of the texture image is mapped to the vertex (0.0, 0.0, -2.0) of the square, (1.0, 1.0) is mapped to (0.0, 1.0, -2.0), (1.0, 1.0) is mapped to (1.0, 1.0, -2.0), and (1.0, 0.0) is mapped to (1.0, 0.0, -2.0).

Instead of using `WAttrib()` with the attribute `texcoord`, the function `Texcoord()` could be used. So the line

```plaintext
WAttrib(w,"texcoord=0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0")
```

could be replaced by

```plaintext
Texcoord(w, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0)
```
A Larger Textures Example  The following more complicated example uses many features of the Unicon 3D graphics facilities described in the previous sections. This example also illustrates the effect of adding texture to a scene. The scene on the left is a scene drawn without any texturing. The scene on the right contains texturing. The scene on the right is a much more realistic scene than the one on the left.

All textures used in the textured scene, except for the unicorn, where captured using a digital camera. These images were then converted into .gif files and scaled to width and height of $2^n$. Directly using an image file is one feature of the Unicon 3D graphics facilities that makes adding textures simpler than using OpenGL.

![Figure 7-10: Untextured and Textured Versions of the Same Scene](image)

```plaintext
procedure main()
    &window := open("textured.icn","gl","bg=black","size=700,700")

    # Draw the floor of the room
    WAttrib("texmode=on", "texture=carpet.gif")
    FillPolygon(-7.0, -0.9, -14.0, -7.0, -7.0, -14.0,
                7.0, -7.0, -14.0, 7.0, -0.9, -14.0, 3.5, 0.8, -14.0)

    # Draw the right and left walls
    WAttrib("texture=wall1.gif", "texcoord=0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0")
    FillPolygon(2.0, 4.0, -8.0, 8.3, 8.0, -16.0, 8.3, -1.2, -16.0, 2.0, 0.4, -8.0)
    WAttrib("texture=wall2.gif")
    FillPolygon(2.0, 4.0, -8.0, -9.0, 8.0, -16.0, -9.0, -1.2, -16.0, 2.0, 0.4, -8.0)

    # Draw a picture
    WAttrib("texture=poster.gif", "texcoord=0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0")
```
FillPolygon(1.0, 1.2, -3.0, 1.0, 0.7, -3.0, 1.2, 0.5, -2.6, 1.2, 1.0, -2.6)
# Draw another picture
WAttrib(“texture=unicorn.gif”, “texcoord=1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)FillPolygon(0.8, 2.0, -9.0, -3.0, -9.0, 3.0, 3.9, -9.0, 0.8, 4.0, -9.0)
# Draw the lamp
WAttrib(“texmode=off”)PushMatrix()Translate(0.7, 0.20, -0.5)Fg(“emission pale weak yellow”)PushMatrix()Rotate(-5.0, 1.0, 0.0, 0.0)Rotate(5.0, 0.0, 0.0, 1.0)DrawCylinder(-0.05, 0.570, -2.0, 0.15, 0.05, 0.17)PopMatrix()Fg(“diffuse grey; emission black”)PushMatrix()Rotate(-5.0, 1.0, 0.0, 0.0)Rotate(6.0, 0.0, 0.0, 1.0)DrawCylinder(0.0, 0.0, -2.5, 0.7, 0.035, 0.035)PopMatrix()PushMatrix()Rotate(6.0, 0.0, 0.0, 1.0)DrawTorus(-0.02, -0.22, -2.5, 0.03, 0.05)PopMatrix()PushMatrix()# Draw the table
WAttrib(“texcoord=auto”, “texmode=on”, “texture=table.gif”)PushMatrix()Rotate(-10.0, 1.0, 0.0, 0.0)DrawCylinder(0.0, 0.2, -2.0, 0.1, 0.3, 0.3)PopMatrix()PushMatrix()Translate(0.0, -0.09, -1.8)Rotate(65.0, 1.0, 0.0, 0.0)DrawDisk(0.0, 0.0, 0.0, 0.0, 0.29)PopMatrix()WAttrib(“texmode=off”, “fg=diffuse weak brown”)PushMatrix()Rotate(-20.0, 1.0, 0.0, 0.0)DrawCylinder(0.0, 0.2, -2.2, 0.3, 0.1, 0.1)PopMatrix()while (e := Event()) ^= “q” do write(image(e), “: “, &x, “, “, &y)end
In order to apply textures to the scene, texturing must be turned on. Next, the texture to be applied is specified. The floor of the scene is drawn using a filled polygon. The default texture coordinates are used to apply the carpet texture to the floor of the room. The tiled appearance on the floor is caused by the use of the default texture coordinates. This can be avoided using user-supplied texture coordinates, as is done for the textures that are applied to the walls and the pictures in the room.

The lamp does not have a texture, so it is necessary to turn off texturing before drawing the lamp. Also for the lamp to be centered properly in the room, transformations are used. Matrices are used to isolate the transformations of the lamp. Finally to draw the table with a textured top and an untextured base, two cylinders and a disk are used. Texturing is applied to a cylinder and the disk. Notice the call

\[
\text{WAttrib(w, "texcoord=auto")}
\]

This resets the texture coordinates to the defaults. Finally, texturing is turned off to draw the base of the table.

**Animation**

Graphics animation is performance sensitive, and Unicon is slower than systems programming languages such as C and C++. Nevertheless, it is possible to write 3D animations in Unicon with acceptable frame rates.

3D animations redraw the entire scene each time an object moves or the user changes point of view. An application can call `EraseArea()` followed by the appropriate graphics primitives to redraw a scene, but the results often appear to flicker. It is better to let Unicon’s runtime system do the redrawing. Unicon maintains a display list of graphics operations to execute whenever the screen must be redrawn; these operations are effectively everything since the last `EraseArea()`. The display list for a window can be obtained by calling `WindowContents()`. The elements of the list are Unicon records and lists containing the string names and parameters of graphics primitives. For example, a call to `DrawSphere(w,x,y,z,r)` returns (and adds to the display list) a record `gl_sphere("DrawSphere",x,y,z,r)`. Instead of redrawing the entire scene to move an object, you can modify its display list record and call `Refresh()`. The following code fragment illustrates animation by causing a ball to slide up and down. In order to *bounce*, the program would need to incorporate physics.

```unicon
sphere := DrawSphere(w, x, y, z, r)
increment := 0.2
every i := 1 to 100 do
  every j := 1 to 100 do {
    sphere.y +:= increment
    Refresh(w)
  }
```
This technique gives animation rates of hundreds of frames per second on midrange PC hardware. Unicon supports smooth animation for a number of objects which varies widely depending on the underlying graphics hardware and software.

Selective rendering and object selection

Many 3D applications model scenes with far more objects than are needed at any particular instant. For example, a virtual building might have many rooms on multiple floors, but only a small fraction is visible from any particular location. The 3D facilities remove objects that are not visible, but doing so becomes too slow for large numbers of objects. An application with a large scene will generally have to perform at least approximate visibility calculations to achieve smooth animation. Such visibility calculations can be performed for each frame, and if the visible objects change, the scene can be re-rendered by rebuilding the display list from scratch. At this point Unicon’s speed can be an issue, as discussed in the previous section.

The function \texttt{WSection()} comes to the rescue. It plays two vital roles. First, it allows portions of the display list to be skipped during rendering, without having to rebuild the display list. Second, it forms the basis for specifying portions of the display list that the user may select (click on) when interacting with the scene. In both cases, calls to \texttt{WSection()} come in pairs, first a call \texttt{WSection(s)} identifies a portion of the display list of interest, then the sequence of 3D calls to render some object or portion of the scene, then a call to \texttt{WSection()} defines the end of that section. Parameter \texttt{s} must be a unique string name or identifier for the section.

The call to create a new section returns a record that contains a field named \texttt{skip}. Setting \texttt{skip} to a non-null value causes the section to be omitted whenever the scene is redrawn. Using \texttt{WSection()} for 3D user input is similar. A program calls \texttt{WAttrib("pick=on")} to turn on 3D selection, after which keyword \texttt{&pick} generates the identifying names for all objects intersected by the ray from the camera through the (x,y) screen location where the mouse was located on the last call to \texttt{Event()}.

7.8 Summary

Graphics are ubiquitous in modern applications. Unicon provides 2D and 3D graphics capabilities that are easy to use, portable building blocks for many programs. The 2D facilities are mature; the 3D interface is new and will evolve. Many elements of the 2D graphics system are used in the 3D graphics interface. Further integration of the 2D and 3D graphics systems is likely in the future.
Chapter 8

Threads

Threads are building blocks for concurrent (also known as parallel) execution. In a concurrent program, the instructions specify multiple things to compute at the same time, during some or most of the program run. On a classic single-processor system these computations will only happen one at a time, but on most modern multiprocessor and multicore systems, the hardware is designed to do several computations at once, and if your program does not ask for as much, it is underutilizing (sometimes severely) the platform.

This chapter describes concurrency in Unicon. It is based on Unicon Technical Report 14; UTR14 on the unicon.org site may amend or supercede this chapter with added features in the future. Threads are an extension of the co-expression type described in Chapter 4 and the system interface described in Chapter 5. Consulting those chapters may be helpful in studying this one.

Concurrent programming introduces techniques, concepts, and difficulties that do not exist in sequential programs. In some situations concurrent programming is a natural way to write programs, such as a server where threads serve different clients. In other situations, concurrent programming improves performance. Long-running programs can run faster by having several threads running cooperatively on several CPU cores, or programs that do a lot of slow I/O operations can allow other non-blocked threads to proceed and utilize the CPU. However, for programs that have a lot of dependencies and are sequential in nature, the complexities of parallelizing them can outweigh the benefits.

This chapter is not a comprehensive concurrent programming guide. It assumes that the reader has some basic knowledge about threads, their programming techniques, and problems, such as synchronization and race conditions. Readers who are unfamiliar with concurrency can refer to a myriad of resources such as [Andr83] or [Bute97] for an overview. Since Unicon’s concurrency facilities are implemented on top of POSIX threads (pthreads), many of the concepts from pthreads programming apply, often with more concise, or higher-level ways of writing things.
8.1 Threads and Co-Expressions

Co-expressions are independent, explicitly sequential execution contexts. Only one co-expression is active at any given moment. When a co-expression is activated, the calling co-expression blocks until the child co-expression returns the execution to it or fails. Threads, on the other hand, can run simultaneously and independently. Threads in Unicon are like special co-expressions that are marked to run asynchronously.

In a concurrent program with two or more threads, each thread has its own program counter, stack pointer, and other CPU registers. However, all of the threads in a program share the address space, open files, and many other pieces of process-wide information. This enables very fast communication and cooperation between threads, which leads to less blocking, faster execution and more efficient use of resources.

Unicon programs start execution in the main() procedure. In a multi-thread programming environment, the procedure main() is the entry point for a special thread referred to as the main thread. This main thread is created by the operating system when the program begins execution. The main thread can create new threads, which can create even more threads. Each thread has an entry point, where it begins executing. Usually this is a procedure but it can be any Unicon expression, as is the case for co-expressions. When a thread first starts running in the entry point, it goes on its own execution path, separate from the thread that created it, which continues to run. A thread never returns. When it ends, it simply terminates; other threads continue to run. An important exception is the main thread; if the main thread ends, the whole program ends. If there are any other threads running, all of them will be terminated.

Since the emergence of the first computer, processors have been increasing in computational power. CPU speeds grew faster than almost all of the other units in the computer, especially the I/O units. This causes programs, especially those which are I/O bound, to spend most of their execution time blocked, waiting for I/O to complete. On systems with multitasking support, several programs run at the same time. When one program blocks for I/O for example, another program is scheduled to run, allowing a better utilization of the system resources. Multitasking offered a way to increase the overall system throughput and boosted the utilization of the increasingly powerful processors. However, multitasking could not help make a process run faster, even on multiprocessor systems.

8.2 First Look at Unicon Threads

Unicon threads facilities give the programmer flexibility in choosing the programming styles that suit the problem at hand. In many situations the same problem can be solved in different ways, using implicit features or explicit ones. The following sections cover the functions and features provided by the thread facilities in Unicon.
8.2. **FIRST LOOK AT UNICON THREADS**

**Thread creation**

Threads can be created in two ways in Unicon, using the **thread** reserved word or using the function **spawn()**. The difference between the two is the separation between creating a thread and running it. The **thread** reserved word creates a thread and starts its execution. The function **spawn()** however, takes a previously created co-expression and turns it into a thread. In many cases the **thread** reserved word allows more concise code. **spawn()** on the other hand is useful in situations where several threads need to be created and initialized before running them. **spawn()** also takes optional parameters to control some aspects of the newly-created thread. The following code creates and runs a hello world thread:

```unicon
thread write("Hello World!")
```

This is equivalent to

```unicon
spawn( create write("Hello World!") )
```

or to

```unicon
c := create write("Hello World!")
spawn(c)
```

Both **thread** and **spawn()** return a reference to the new thread. The following program creates 10 threads:

```unicon
procedure main()
  every i := 10 do thread write("Hello world! I am thread: ", i )
  write("main: done")
end
```

In this example, the main thread continues to execute normally after firing 10 threads. Because of the non-deterministic nature of threads, there is no guarantee which thread gets to print out its "hello world" message first, or in what order the messages are printed out, including the message from the main thread "**main: done**". All of the possible permutations are valid. No assumptions can be made about which thread will continue running or finish first. It depends on the host OS CPU process/thread scheduler. The order is unpredictable.

Furthermore, the main thread might finish and terminate the program before some or all of the threads get executed or print out messages. To avoid such situations, the main thread needs to wait for other threads to finish before exiting the program. This is can be achieved by using the function **wait()**, which blocks the calling thread until the target thread is done. The above program can be rewritten as follows:

```unicon
procedure main()
  L := []
```

every i := !10 do put(L, thread write("Hello world! I am thread: " , i))
every wait(!L)
write("main: done")
end

wait(!L) tells the main the thread to wait for every thread to finish, causing the message "main: done" to be the last thing printed out before the program ends. wait() is useful in cases where threads need to synchronize so that one thread blocks until another finishes. wait() provides a very basic synchronization technique, but most concurrent programming tasks need more synchronization than waiting for a thread to finish. Advanced synchronization mechanisms are discussed below.

Thread evaluation context

Similar to co-expressions, threads have their own stack, starting from a snapshot of parameters and local variables at creation time. This allows co-expressions and threads to be used outside the scope where they are created. It also allows a thread to start by using the variable values at the time of its creation, rather than when running it in the case of spawn(). An important side effect of this process is avoiding race conditions, because each thread gets a copy of the variables instead of having all the threads competing over the same shared variables. Race conditions and thread-safe data will be covered in depth in the following section. The following example and its output demonstrate the idea of an evaluation context:

procedure main()
    local x:= 10, y:=20, z:=0
    write( "Main thread: x=" , x, ", y=" , y, ", z=" , z)
    thread (x:=100) & write("Thread 1: x=" , x)
    thread (y:=200) & write("Thread 2: y=" , y)
    thread (z:=x+y) & write("Thread 3: z=" , z)
    delay(1000)
    write( "Main thread: x=" , x, ", y=" , y, ", z=" , z)
end

The output is:

Main thread: x=10, y=20, z=0
Thread 3: z=30
Thread 1: x=100
Thread 2: y=200
Main thread: x=10, y=20, z=0
The delay(1000) should give the threads enough time to finish before the main program finishes. This should not be left to chance: wait() will block until the threads finish, instead of a 1 sec delay.

The output shows that the changes to the variables are per-thread, and not visible in the main thread or in the other threads. The copies of local variables in different threads can be thought of as passing parameters by value to a procedure. This is true for local variables of immutable data types; on the other hand, global variables and mutable types, such as lists, are shared. Any change in the structure of such types is visible across all threads. Contrast the following example with the one above:

```unicon
procedure main()
    local L
    L := [20, 10, 0]
    thread (L[1]:=100) & write("Thread 1: L[1]=", L[1])
    delay(1000)
end
```

with output

```
Thread 3: L[3]=300
Thread 1: L[1]=100
```

Instead of using 3 variables x, y, and z, a list of size 3 is used. x from the previous example maps to L[1], y to L[2], and z to L[3]. The program does the same thing as before, but any change to the content of L is visible in other threads. Unlike the output in the first case, where the values of x, y, and z remained the same in the main thread, this output shows that the changes to the list elements in the other threads were visible in the main thread.

### Passing arguments to threads

When creating a new thread for a procedure, the parameters that are passed to the procedure at creation time can be thought of as a one-time one-way communication between the creator thread and the new thread. This is very useful in initializing the new thread or passing any data that the thread is supposed to work on. The following program has 3 threads in addition to the main thread. The main thread passes a list to each “worker” thread, and each worker sums the list and prints the sum to the screen:
procedure main()
  L1 := [1, 2, 3]
  L2 := [4, 5, 6]
  L3 := [7, 7, 9]
  t1 := thread sumlist(1, L1)
  t2 := thread sumlist(2, L2)
  t3 := thread sumlist(3, L3)
  every wait(t1|t2|t3)
end

procedure sumlist(id, L)
  s := 0
  every s +:= !L
  write(" Thread id=", id, ", result=", s)
end

The output is

Thread id=2, result=15
Thread id=1, result=6
Thread id=3, result=23

Since the lists are independent, there is no possibility of a race condition. The example shows that the second thread was the first to finish and print its result. If the problem solution requires sharing data or guaranteeing that one thread should finish before another, then a synchronization mechanism should be used. This is the subject of the next section.

### 8.3 Thread Synchronization

Thread synchronization can be done in many different ways. Some problems require more synchronization than others. Some may require advanced synchronization mechanisms and rely on the language support to achieve full control over the execution of threads and protect shared data. This section covers many synchronization techniques in Unicon, introduces the concept of race condition in a multi-thread environment, and the concept of thread-safe code.

**The non-deterministic behavior of threads**

Programming with threads introduces a whole new set of concepts and challenges that non-threaded programs do not have to deal with. In most multi-threaded programs, threads need to communicate through shared data. Because threads run in a non-deterministic
order, they access and update shared data in a non-deterministic fashion. Consider the following popular example where two threads, $T_1$ and $T_2$, try to increment a shared variable $x$ whose initial value is 0.

\[ T_1 \quad T_2 \]
\[ x := x+1 \quad x := x+1 \]

While $x:=x+1$ may not look like it could cause a problem, in reality it does because it is not atomic. In many computer systems, it can be broken down into three operations: fetch the value of $x$, add 1 to it, and store the new value back in $x$. These three operations might occur at different times in different threads. Thread $T_2$ for example might fetch $x$, followed by $T_1$ also fetching it, but before $T_2$ stores back the new value of $x$, leaving $T_1$ working on the old value of $x$. $T_1$ should not be allowed to read the value of $x$ while another thread, such as $T_2$, is updating it. Consider the following scenarios, starting with $x:=0$:

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>fetch $x$ (0)</td>
<td>fetch $x$ (0)</td>
</tr>
<tr>
<td>increment $x$ (1)</td>
<td>increment $x$ (1)</td>
</tr>
<tr>
<td>store $x$ (1)</td>
<td>store $x$ (1)</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>fetch $x$ (0)</td>
<td>fetch $x$ (0)</td>
</tr>
<tr>
<td>increment $x$ (1)</td>
<td>increment $x$ (1)</td>
</tr>
<tr>
<td>store $x$ (1)</td>
<td>store $x$ (1)</td>
</tr>
</tbody>
</table>

The final value of $x$ is 1.

The final value of $x$ is 2.

In scenario 1 the final value of $x$ is 1, even though there are two increments done by the two threads. In scenario 2 however the final value is 2. This outcome is not necessarily a problem, or a bug that must be fixed. Non-deterministic execution is a part of multi-threaded programming that many programs can live with. For example, if one or more threads depend on a counter to update the screen every 100 increments or so, but this number does not need to be exactly 100, then the threads can increment the counter without worrying about races and about synchronizing access to the shared counter. If deterministic execution must be guaranteed, programmers have to take extra steps to ensure a specific order and predictable results. That is where thread synchronization comes into play.

**User-defined synchronization**

For some simple situations, synchronization can be achieved without relying on special primitives provided by the language. For example, if one thread is waiting for another to finish a task, a shared flag variable can be used. In the following example, the main thread might finish before the child thread:
As seen in the previous section, this can be handled using `wait()` or `delay()`. The `wait()` function is the best solution for this situation. `delay()` also works but there are two problems associated with it: it forces the program to wait a lot longer than necessary, and second, if the delay time is not long enough, depending on the system, the main thread might still finish before the child thread. Actually even with a long delay, there is no guarantee the child thread will finish first. In a real application, `delay()` would be a poor choice. Finally, here is an alternative solution that does not use `wait()`:

```
global done
procedure main()
    thread (write("I am a thread: Hello world!") & done := "true")
    until \done
end
```

In this case, the loop `until \done` ensures that the main thread keeps spinning until the child thread set the variable `done` to a non-null value. It avoids the problems with using `delay()`, at the expense of fully occupying one CPU in a spin-lock. Note that declaring `done` to be global is key. If `done` were local, the main thread would spin indefinitely because any change to `done` in the child thread would be invisible in the main thread.

If none of these approaches seems acceptable, that is a good sign. Use techniques from the following sections to avoid such inefficient synchronization.

### Language support for synchronization

Using function `wait()` or global variables to synchronize threads might be sufficient in some situations, but most problems require the more efficient synchronization made possible by mutexes and condition variables.

#### Critical regions and mutexes

A **mutex** (from mutual exclusion) is a synchronization construct used to protect shared data and serialize threads in **critical regions**, sequences of instructions in which only one thread may execute at a time or an error will occur. In the example discussed at the beginning of this chapter, two threads compete to increment the variable `x`. The end result might not be what the programmer intended. In such cases a mutex may be used to protect access to the variable `x`. Any operation, including data structure traversal, where more than one thread can execute non-deterministically, and which may lead to data corruption or incorrect results is called **thread-unsafe**. Thread-unsafe code or data structures have to be handled correctly via synchronizations mechanism to achieve correct behavior and results.
8.3. THREAD SYNCHRONIZATION

A mutex object is created using the `mutex()` function. The returned object can be locked/unlocked via the functions `lock()` and `unlock()` to serialize execution in a critical region. The following example demonstrates the use of a mutex to protect increments to the global variable `x`:

```plaintext
global x
procedure main()
   mtx_x := mutex()
   x := 0
   t1 := thread inc_x(mtx_x)
   t2 := thread inc_x(mtx_x)
   every wait(t1 | t2)
   write("x=", x)
end

procedure inc_x(region)
   lock(region)
   x := x + 1
   unlock(region)
end
```

It is important to note that the mutex object has to be initialized only once and can then be shared between all threads (here `t1` and `t2`) accessing the critical region (`x := x + 1`). `lock(region)` marks the beginning of the critical region protected by the mutex, and `unlock(region)` marks its end. When a thread calls `lock(region)`, it tries to acquire the mutex `region`. If `region` is not “owned” by any other thread, `lock(region)` succeeds, the thread becomes the owner of the mutex `region`, and then enters the critical region. Otherwise the thread blocks until the current owner of the mutex leaves the critical region by calling `unlock(region)`. Since there are two threads and `x := x+1` is protected by a mutex, the output of the program is guaranteed to be `x=2`, unlike the case where a mutex is not used, and where `x=1` or `x=2` are possible outputs.

The more critical-regions/mutexes a concurrent program has, the slower it runs. The length of the critical region also affects the performance. The longer the critical region, the more time it takes a thread to traverse it and release the mutex, which increases the probability that other threads become blocked waiting to acquire the mutex and enter the critical region. Locking a mutex and forgetting to unlock it is very likely to lead to a deadlock, a common problem in concurrent programming, where all threads block waiting for each other, and for resources to become available. Because all threads are blocked, resources will not be freed, and the block persists indefinitely.

Unicon provides a special syntax for critical regions equivalent to a `lock()/unlock()` pair, that aims mainly to guarantee that a mutex is released at the end of a critical region, besides enhancing the readability of the program. Here is the syntax:
critical mtx: expr

This is equivalent to:

lock(mtx)
expr
unlock(mtx)

Given a global variable named region that has been initialized as a mutex, the code to increment x in the previous example can be written as:

critical region: x := x + 1

The critical region syntax only unlocks the mutex if it executes to the end. If there is a return or break in the region’s body, it is the programmer’s responsibility to explicitly unlock the mutex. For example:

critical region: {
    if x > 100 then { unlock(region); return }
    x := x + 1
}

In some situations, a thread might have several tasks to finish and may not want to block waiting for a mutex that is locked by another thread. For example, if a thread is creating items that can be inserted in one of several shared queues, the thread can insert every new item in the first queue that it acquires. trylock() is an alternative non-blocking function for locking. If the thread cannot acquire the mutex immediately, the function fails. The most suitable way to use trylock() is to combine it with an if statement, where the then body unlocks the mutex after finishing the work on the protected object, as follows:

if trylock(mtx) then {
    expr
    unlock(mtx)
}

Both lock() and trylock() return a reference to the mutex or the object they acquired (upon succeeding in case of trylock()). This makes it very convenient to write code like the following, assuming L1 and L2 are lists that are both marked as shared:

item := newitem()
if L := trylock(L1 | L2) then {
    put(L, item)
    unlock(L)
}
8.3. THREAD SYNCHRONIZATION

Note that trylock() may fail to lock any of the lists, leaving item unprocessed. Depending on what the code needs to do, if it is required to guarantee that it does not proceed before one of the locks to L1 or L2 succeeds, then it can be written as follows:

\[
\begin{align*}
\text{item} & := \text{newitem()} \\
\text{until L} & := \text{trylock(L1 | L2)} \\
\text{put}(L, \text{item}) \\
\text{unlock}(L)
\end{align*}
\]

Initial clause A procedure in a Unicon program can have an initialization clause at its top. This gets executed only once the first time the procedure is entered. The initial clause provides a very convenient way to place local static variables and their initialization in the same procedure, instead of relying on global variables and having to initialize them somewhere else. A procedure that produces a sequence of numbers one at each call can be written as:

\[
\begin{align*}
\text{procedure seq()}
\end{align*}
\]

\[
\begin{align*}
\text{static i} \\
\text{initial i := 0} \\
\text{i := i + 1} \\
\text{return i}
\end{align*}
\]

Initial clauses are thread-safe. They can be thought of as a built-in critical region that is run only once. No thread is allowed to enter the procedure if there is a thread still executing in the initial block. This can be useful in a concurrent environment to do critical initialization, such as creating a new mutex object instead of declaring a mutex variable to be global and initializing it somewhere else, or passing it from one function to another where it will be actually used. A concurrent version of seq() would look like this:

\[
\begin{align*}
\text{procedure seq()}
\end{align*}
\]

\[
\begin{align*}
\text{local n} \\
\text{static i, region} \\
\text{initial \{ i := 0; region := mutex() \}} \\
\text{critical region: n := i := i+1} \\
\text{return n}
\end{align*}
\]

With the use of the initial clause, seq() is self-contained and thread-safe. Note the use of the local variable n to temporarily hold the value of the counter i while still in the critical region. That is because once the thread leaves the critical region, there is no guarantee that the value of i would remain the same before it is returned. Using the variable n guarantees that the value returned is correct, even if the value of i is changed by another thread.
CHAPTER 8. THREADS

Thread-safe data structures  In Unicon, mutexes are not just independent objects as described above, they are also attributes of other objects, namely attributes of the mutable data types. Any data structure in Unicon that can be used in a thread-unsafe manner can be protected by turning on its mutex attribute. Instead of declaring a separate mutex and locking and unlocking it, the structure can just be marked as “needs a mutex/protection” and the language does an implicit locking/unlocking, protecting the operations that might affect the integrity of the structure. For example, if several threads are pushing and popping elements into and out of a list, these are thread-unsafe operations that require protection.

The value of implicit mutexes is made clear after considering the alternative. The following producer-consumer example uses a list to send and receive data, and protects it using an explicit mutex:

```unicon
procedure main()
    L := []
    mtx := mutex()
    p := thread produce(L, mtx)
    c := thread consume(L, mtx)
    every wait(p | c)
end

procedure produce(L, region)
    every i := !10 do
        critical region: put(L, i)
    end
end

procedure consume(L, region)
    i := 0
    while i < 10 do
        critical region: if x := get(L) then i += 1 & write(x)
    end
end
```

Using a thread-safe list results in fewer lines of code, and in a more efficient program doing less locking and unlocking at the language level, or even not doing explicit locking at all. For example, the above program may be rewritten as:

```unicon
procedure main()
    L := mutex([ ])
    p := thread produce(L)
    c := thread consume(L)
    every wait(p | c)
end

procedure produce(L)
    every put(L, !10)
end
```
procedure consume(L)
  i := 0
  while i < 10 do
    if x := get(L) then i := i + 1 & write(x)
  end

The `produce()` and `consume()` procedures do not do any locking, making concurrent programming in such a case just as easy as writing a sequential program. It is only necessary to notify the language at the beginning that the data structure is shared, by passing it to the `mutex()` function. This function takes a second optional parameter denoting an existing mutex object or an object that is already marked as shared (has a mutex attribute). Instead of creating a new mutex object for the data structure, the existing mutex is then used as an attribute for the data structure. If the second object is a structure that is not marked as shared, a new mutex is created. This is useful when two objects need to be protected by the same mutex. For example, the list `L` and the table `T` in the following example share the same mutex:

```plaintext
mtx := mutex()
L := mutex([], mtx)
T := mutex(table(), mtx)
```

which is equivalent to the following if the mutex does not need to be explicit:

```plaintext
L := mutex([])
T := mutex(table(0), L)
```

or

```plaintext
L := []
T := mutex(table(0), L)
```

In all cases, `lock(L)` and `lock(T)` lock the same mutex, serializing execution on both data structures. Not all operations on data structures produce correct results, only “atomic” operations do. In other words, implicit locking/unlocking takes place per operation, which means that even if each of the two operations is safe, the combination might not be. A critical region is still needed to combine the two. For example, if `L[1]` has the value 3 and two threads are trying to increment `L[1]`:

```plaintext
```

the resulting `L[1]` could be 4 or 5. That is because reading `L[1]` (the right side of the assignment) and storing the result back in `L[1]` are two different non-atomic operations, separated in time. The good news is that solving such an issue does not require an extra explicit mutex. If `L` is marked as shared (passed to the `mutex()` function) it can be passed to `lock()`/`unlock()` functions. It can be used with the critical syntax like this:

```plaintext
```
**Condition variables**  Mutexes are used to protect shared data in critical regions, and block threads if there is more than one thread trying to enter the region. *Condition* variables take thread blocking/resumption to a new level that is not tied to accessing shared data like a mutex. A condition variable allows a thread to block until an event happens or a condition is satisfied. For example, the previous section showed a producer/consumer problem where the consumer keeps spinning to get values out of the shared list. In real-life applications, any spinning could be a waste of resources; other threads, including producer threads could be using the resources to do something useful instead. The consumer needs to block until there is data to process in the list. This is where a condition variable comes into play. A condition variable is created using the function `condvar()`. The returned object is a condition variable that can be used with `wait()` and `signal()` functions. `wait(cv)` blocks the current thread on the condition variable `cv`. The thread remains blocked until another thread does a `signal(cv)`, which wakes up one thread blocked on `cv`. A very important aspect of using a condition variable is that the variable must always be associated with a mutex. More specifically, the `wait()` function has to be always protected by a mutex. Unicon provides a built-in mutex for condition variables which can be thought of as an attribute similar to thread-safe data structures. This means that a condition variable can also be used with `lock()/unlock()` functions or the `critical` clause. It is important to realize that not only `wait()` has to be protected by a critical region, but also the condition or the test that leads a thread to wait on a condition variable. See the following example:

```unicon
if x=0 then wait(cv)
```

A thread wants to wait on `cv` if `x=0`, but what happens if the value of `x` has changed between the test and the call to `wait(cv)`? If a second thread changes the value of `x` and signals `cv` to wake up the first thread, while the first thread transitions from the test to `wait()`, it may miss the wake up signal and might block indefinitely because it is waiting on a condition variable that it should not wait on. The correct way to use `wait()` with a condition variable is

```unicon
lock(cv)
if x=0 then wait(cv)
unlock(cv)
```

or:

```unicon
critical cv: if x=0 then wait(cv)
```

Because other threads might need to access the condition variable while some threads are waiting on it, the `wait()` function atomically blocks the thread and releases its corresponding mutex. After receiving a wake-up signal, the blocked thread wakes up, acquires the mutex (blocking if necessary) and continues executing, and that is when `wait()` returns. It is good
practice to do the condition variable test before assuming that it is in one state or another. This leads to a more correct way to use condition variables that ensures that a thread does not leave \texttt{wait()} before guaranteeing the test is in a specific state, as follows:

\begin{verbatim}
critical cv: while x=0 do wait(cv)
\end{verbatim}

Using a \texttt{while} in place of \texttt{if} will ensure that the thread goes back to sleep if it happens to wake up and the condition has not changed.

The producer/consumer example mentioned above can be rewritten using a condition variable. Since the consumer needs to sleep/wake up depending on the availability of elements in the list, the state of the list must be guaranteed to remain the same while interacting with the condition variable for the reason explained above (missing wake-up signals). The list and the condition variable have to be protected by the same mutex. \texttt{condvar()} allows an optional argument, an existing mutex that is to be associated with the condition variable. In the original example, using an independent mutex to protect the condition variable looks like this:

\begin{verbatim}
procedure main()
  L := []
  mtx := mutex()
  cv := condvar(mtx)
  p := thread produce(L, cv)
  c := thread consume(L, cv)
  every wait(p | c)
end

procedure produce(L, cv)
  every i := 1 to 10 do {
    critical cv: put(L, i)
    if *L=1 then signal(cv)
  }
end

procedure consume(L, cv)
  i := 0
  while i < 10 do {
    if *L=0 then critical cv: until *L>0 do wait(cv)
    if x := get(L) then i +:= 1 & write(x)
  }
end
\end{verbatim}

Another way to write this program is on top of the thread-safe list example. Since there is no explicit mutex to pass to the \texttt{condvar()} function in the original example, a mutex can be first created and then passed along with the list to the \texttt{mutex()} function. The same
mutex then can be passed to \texttt{condvar()}. The function binds the mutex already associated with the list to the condition variable. The final result is the same as in the explicit mutex example, a list and a condition variable sharing the same mutex. Here is the example again with the shared list and a condition variable:

\begin{verbatim}
procedure main()
  mtx := mutex()
  L := mutex([], mtx)
  cv := condvar(mtx)
  p := thread produce(L, cv)
  c := thread consume(L, cv)
  every wait(p | c)
end

procedure produce(L, cv)
  every put(L, !10) & *L=1 & signal(cv)
end

procedure consume(L, cv)
  i := 0
  while i < 10 do
    if x := get(L) then
      i += 1 & write(x)
    else
      critical cv: until *L>0 do wait(cv)
  end
end
\end{verbatim}

In previous examples, calls to \texttt{signal()} are not protected by any mutex. \texttt{signal()} does not need protection, because it does not block the thread, and there are no worries about a deadlock. The worst thing that can happen is signaling a condition variable that does not have any thread waiting on it, which is not a problem. However protecting calls to \texttt{signal()} is not an issue either. Depending on the problem, doing it one way or the other might be more or less efficient. There is no need to have a thread that spends a lot of time locking/unlocking a mutex if it is not necessary, creating contention in the critical region. But a thread that keeps wasting time signaling condition variables that have no threads waiting on them is also undesirable.

The \texttt{signal()} function takes a second optional parameter: the number of threads to be woken up. The default is one, but it can be any positive value. For example:

\begin{verbatim}
every !4 do signal(cv)
\end{verbatim}

can be written as:

\begin{verbatim}
signal(cv, 4)
\end{verbatim}
Furthermore, if all of the threads waiting on \texttt{cv} need to be woken up, a special 0 (or \texttt{CV\_BROADCAST}) value can be passed to \texttt{signal()}, causing it to broadcast a wakeup call for all threads waiting on \texttt{cv}:

\begin{verbatim}
signal(cv, 0)
\end{verbatim}

or

\begin{verbatim}
signal(cv, CV\_BROADCAST)
\end{verbatim}

8.4 Thread Communication

Traditionally, co-expressions communicate implicitly, or explicitly using the \texttt{@} operator. All co-expression communication is synchronous; the calling co-expression is blocked and the called co-expression runs. This simple communication model is called \textit{activation} in Unicon. A co-expression \texttt{C1} can activate another co-expression \texttt{C2} using the syntax \texttt{x@C2}, where \texttt{x} is an optional value to be transmitted from \texttt{C1} to \texttt{C2}. \texttt{C1} waits until it gets activated by \texttt{C2} or any other co-expression directly or indirectly activated by \texttt{C2}. As mentioned earlier, implicit activation takes place whenever a co-expression produces a value or falls off its end. With implicit activation, the co-expression activates its parent (the last co-expression to activate it).

Threads take co-expression communication to a new level with their dynamic nature. Threads run concurrently; in many cases, a running thread just wants to send a value to another thread without waiting for a reply, or receive a value from another thread, if there is one, without waiting. The \texttt{@} operator is not suitable for this kind of (asynchronous) communication. Unicon adds four operators dedicated to asynchronous communication. These are \texttt{@>}, \texttt{@>>}, \texttt{<@} and \texttt{<<@}. The operators correspond to send, blocking send, receive, and blocking receive.

Thread messaging queues

Before exploring how these communication operators are used, look at messaging queues and how they are utilized to support communication between threads. Each thread maintains two queues called the \textit{inbox} and \textit{outbox} that are created with the thread. When a thread sends a message with an explicit destination, the message is queued in the destination’s inbox. Otherwise, it is queued into the sender’s outbox. A thread can receive messages from another thread by dequeuing messages from the source’s outbox if there is an explicit source, otherwise it dequeues messages from its own inbox. Figure 8-1 presents two threads with inboxes and outboxes.
send and receive operators

The @> (send) and <@ (receive) operators communicate messages containing arbitrary data between threads. The operators support co-expressions as well, with the same semantics. The send operator has the syntax

\[ x @> T \]

where \( x \) can be any data type, including null, which is equivalent to omitting it. \( T \) refers to a thread to which \( x \) is transmitted: \( x \) is placed in \( T \)'s inbox. \( x \) can be picked by \( T \) using the receive operator which is presented later. The send operator can also have no destination, as in

\[ x @> \]

In this case \( x \) is sent to no one, instead it is placed in the sender’s outbox. The operator can be read as “produce \( x \)”. \( x \) can then be picked up later by any thread wanting a value from this sender. For example the sender in this case might be a creating prime numbers and placing them in its outbox to be ready for other threads.

The receive operator is symmetric to send, and takes two forms, with explicit source or with no source, as follows:

\[ <@ T \]
\[ <@ \]

The first case reads “receive a value from \( T \)”; it obtains a value from \( T \)'s outbox. In the prime number example mentioned above, \(<@ T \) would be the way to get a prime number produced by \( T \). \(<@ \) on the other hand reads values directly from the receiver’s inbox. It reads messages sent explicitly to the thread doing the receive operation.

Both @> and <@ can succeed or fail. In the case of <@ the operator succeeds and returns a value from the corresponding queue (inbox/outbox) depending on the operand if the queue is not empty. If the queue is empty the operation fails directly. In the case of @>, if the value is placed in the corresponding queue the operation succeeds and returns
the size of the queue. If the queue is full, the send operation fails. The inbox/outbox for each thread is initialized to have a limited size (it can hold up to 1024 values by default). This limit can be increased or decreased depending on the application needs. The limits are useful so that queue sizes do not explode quickly by default. They also provide an implicit communication/synchronization as explained later in following sections. Let us look at the producer/consumer example again written using the new operators:

```plaintext
procedure main()
p := thread produce()
c := thread consume(p)
every wait(p | c)
end

procedure produce()
every !10 @> # place values in my outbox
end

procedure consume(p)
i := 0
while i < 10 do
  if x := <@ p then # get values from p
    i +:= 1 & write(x)
  end
end
```

Each thread has exactly one inbox and one outbox, and each operator call is mapped to only one of these inboxes or outboxes as seen in Figure 1. All messages from all threads coming to thread B in the figure end up in its inbox. All threads trying to receive messages from A compete on A’s outbox. Both the inbox and the outbox queues are public communications channels, and it is impossible to distinguish the source of a message if there are several threads sending messages to the same thread at the same time. Furthermore, if `<@` has an explicit source like A in Figure 1, it only looks in A’s outbox, and does not see messages from A coming directly to the inbox. Applications that require the sender’s address can attach that information to messages by building them as records with two fields, one field for data and the other containing the sender’s address. A better approach for private communications for some applications is the use of lists shared between the two communicating threads or the use of private communication channels discussed later in this document.

### Inbox/Outbox and the `Attrib()` function

As seen in previous sections, communication between threads is done through inbox/outbox queues which have size limits. The size limit, which defaults to 1024, and the actual size dictate how synchronization is merged with the communication. The size operator `*` can
be used with a thread to query its actual outbox size (how many values it contains, not the maximum limit), as follows:

\[
\text{outbox\_size} := *T
\]

But this is only a single attribute for one queue. To access or change other queue attributes, a new function `Attrib()` is introduced. This function uses the form `Attrib(handle, attribcode, value, attribcode, value, ...)`. The integer codes used by this function are defined in an include file `threadh.icn`. This header file is part of the `threads` package, which can be used by a program via

```
import threads
```

When values are omitted, `Attrib()` generally returns attribute values. To get the size of the outbox (the same as the `*` operator), the code is

```
outbox\_size := Attrib(T, OUTBOX\_SIZE)
```

similarly,

```
inbox\_size := Attrib(T, INBOX\_SIZE)
```

gets the current size of the inbox. On the other hand

```
Attrib(T, INBOX\_LIMIT, 64, OUTBOX\_LIMIT, 32)
```

sets the inbox and outbox size limits to 64 and 32 respectively. The following table summarizes the available attributes and their meanings.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
<th>Read/Write?</th>
</tr>
</thead>
<tbody>
<tr>
<td>INBOX_SIZE</td>
<td>Number of items in the inbox</td>
<td>Read Only</td>
</tr>
<tr>
<td>OUTBOX_SIZE</td>
<td>Number of items in the outbox</td>
<td>Read Only</td>
</tr>
<tr>
<td>INBOX_LIMIT</td>
<td>The maximum number of items allowed in the inbox</td>
<td>Read/Write</td>
</tr>
<tr>
<td>OUTBOX_LIMIT</td>
<td>The maximum number of items allowed in the outbox</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>
8.4. THREAD COMMUNICATION

Blocking send and receive

In many situations senders and receivers generate and consume messages at different speeds or based on needs. Instead of overloading slow receivers or busy waiting for slow senders, the two ends of the communication need a synchronizing mechanism to tell them when to send new messages or when a new message is available. Two more send and receive operators provide such functionality, the blocking send operator @>> and the blocking receive operator @$@. These can be used in the same way as @> and <@, except that instead of failing when the operation cannot be completed, the new operators block and wait until the operation succeeds. In the simple producer/consumer example, the producer is only producing 10 values, since the default size of the queue is 1024, using a blocking send would not make any difference. The consumer however, can make use of the blocking receive instead of spinning in some cases while the queue is empty and the original blocking receive just keeps failing. Take a closer look at the consumer code again:

```
procedure consume(p)
  i := 0
  while i < 10 do
    if x := <@ p then # get values from p
      i += 1 & write(x)
    end
  end
```

Using an if statement with <*> checks whether the operation succeeds in receiving a value. A blocking receive is more suitable in this case and it simplifies the loop slightly, since the if can be dropped, and also the counter is not necessary anymore. The counter was previously necessary because the loop needs to keep track of how many <*> were needed to count to 10. The consumer can be rewritten as

```
procedure consume(p)
  # get exactly 10 values from p, block if necessary
  every !10 do write($@$ p)
end
```

In some cases, a thread might want to use a blocking receive to get values from a second thread, but it is not willing to block indefinitely; it may do some other useful work instead of waiting. The $@$ operator accepts a timeout parameter to impose a limit on how long to wait for a result before giving up. Here is how $@$ would look in this case:

```
result := timeout @$ # get from my inbox
```

or

```
result := timeout @$ T # get from T's outbox
```
The timeout operand is a non-negative integer denoting the maximum time to wait in milliseconds. Negative integers are treated as a null value, defaulting to an indefinite blocking receive. A 0 operand indicates no waiting time, effectively resulting in a non-blocking receive. The following table summarizes the different forms of the send and receive operators and their operands:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>@&gt;</td>
<td>msg@&gt;</td>
<td>Place msg in my outbox, fail if the outbox is full</td>
</tr>
<tr>
<td>(send)</td>
<td>msg@T</td>
<td>Place msg in T’s inbox, fail if T’s inbox is full</td>
</tr>
<tr>
<td>&lt;@</td>
<td>@</td>
<td>Get a message from my inbox, fail if the inbox is empty</td>
</tr>
<tr>
<td>(receive)</td>
<td>@T</td>
<td>Get a message from T’s outbox, fail if T’s outbox is empty</td>
</tr>
<tr>
<td>@&gt;&gt;</td>
<td>msg@&gt;&gt;</td>
<td>Place msg in my outbox, block if the outbox is full</td>
</tr>
<tr>
<td>(blocking send)</td>
<td>msg@&gt;&gt;T</td>
<td>Place msg in my T’s inbox, block if the T’s inbox is full</td>
</tr>
<tr>
<td>&lt;&lt;@</td>
<td>&lt;&lt;@</td>
<td>Get a message from my inbox, block if the inbox is empty</td>
</tr>
<tr>
<td>(blocking receive)</td>
<td>&lt;&lt;@T</td>
<td>Get a message from T’s outbox, block if it is empty</td>
</tr>
<tr>
<td>&lt;&lt;@</td>
<td>n&lt;&lt;@</td>
<td>Get a message from my inbox, block up to n milliseconds waiting for an inbox message to become available</td>
</tr>
<tr>
<td></td>
<td>n&lt;&lt;@T</td>
<td>Get a message from T’s outbox, block up to n milliseconds waiting for a message to become available there</td>
</tr>
</tbody>
</table>

Most applications use only a few of these modes. In a fast sender/slow receiver application, the sender would block when the queue is full and unblock when the queue is empty (using @>>). The receiver would consume messages from the queue until it is empty, and then block until there is a new message added to the queue (<<@). For some applications however this communication scheme might not be optimal, hence many options are provided. The different options in the table above give the programmer a wide range of control over when to block or resume a thread based on the availability of data in the communication queues. This control covers the needs of many applications and provides simple ways to abstract concurrent programming activities such as load balancing and efficient use of resources.

**Private communication channels**

As mentioned in the previous sections, inbox and outbox communication queues are visible by all threads all the time. In some scenarios two or more threads need to communicate
with each other without worrying about other threads sending and receiving messages at the same shared queues. While it is possible to build a protocol at the application level on top of the inbox and outbox queues to achieve such behavior, it is simpler and more efficient to have the threads communicate privately. This kind of communication can be done by sharing a list between two threads and protecting it by an explicit mutex, or using a thread-safe list. A more formal way for such communication is to use the \texttt{channel()} function.

Starting a private communication is similar to a network connection, except that this connection is taking place between two threads in the same process instead of two different processes that may be on different machines. A private communication channel between two threads can be created using the library procedure \texttt{channel()}.

\texttt{channel()} is part of the \texttt{threads} package, so \texttt{import threads} is necessary to use it. It takes one parameter, which is the thread with which the connection will be initiated. If \texttt{channel()} succeeds, it returns a list representing a communication channel between the two threads. Representing a bidirectional channel that can be used by the two threads, given that each thread calls the function \texttt{channel()} with the other thread as an argument. Here is an example.

In thread A:

\begin{verbatim}
chB := channel(B) | "failed to open a channel with B"
\end{verbatim}

In thread B:

\begin{verbatim}
chA := channel(A) | "failed to open a channel with A"
\end{verbatim}

A channel is a \textit{directional} communication medium. One thread should use it as an outbox, and the other should use it as an inbox; only one thread will send messages over the channel while the other receives them from the other end. The provided channel can be used with the communication operators (all four of them) with the same semantics as before. The only difference in this case is that the right operand is a communication channel instead of a thread. In the channel example below, the main thread transmits the consumer’s identity to the producer (\texttt{c @> p}), who receives it via \texttt{c := <<@}}:

\begin{verbatim}
import threads
procedure main()
  p := thread produce()
  c := thread consume(p)
  c @> p
  every wait(p | c)
end

procedure produce()
  c := <<@
\end{verbatim}
```plaintext
chC := channel(c)
every !10 @> chC  # place values in channel c
end

procedure consume(p)
  chP := channel(p)
  every !10 do write(<<@chP)
end
```

### 8.5 Summary

True concurrency opens up major new application domains for the Unicon language. More importantly, it enables the language to utilize more than the small fraction of modern processors utilized by traditional sequential execution. For example, on a typical quad-core desktop, many applications will be able to get between 2× and 4× the performance of a sequential Unicon program with relatively minor changes. This is comparable to the speedup typically delivered by the optimizing compiler. Some applications will be able to do even better on processors with more cores.

This document presented Unicon’s concurrency facilities from a programmer’s perspective. The implementation and its performance are described in more detail in [Al-G12]. There are major areas for future work, including GPU- and APU support, and various forms of implicit concurrency that can be added to the language.
Chapter 9

Execution Monitoring

Unicon’s execution monitoring facilities allow the user to execute a Unicon program under the observation of one or more monitoring programs, also written in Unicon. This chapter presents the monitoring architecture and a standard execution monitoring scenario, followed by details of the language features involved. This chapter is based on “Program Monitoring and Visualization” [Jeff99], which has many additional examples.

9.1 Monitor Architecture

The monitoring facilities components are summarized in Figure 9-1. Many of these components are general-purpose language features that are useful independent of execution monitoring.

Figure 9-1: The Alamo architecture
Monitor Terminology

The terminology used in discussing Unicon’s execution monitoring architecture relates to events and the linguistic features associated with them.

**Dynamic loading** — The ability to load multiple programs into a shared execution environment supports monitor access to target program data. *Dynamic linking* is not desirable in the context of execution monitoring; the names in the monitor are distinct from those in the target program.

**Synchronous execution** — The monitor and target program execute independently, but not concurrently. This allows the monitor to control target program execution using a simple programming model. Unicon’s *co-expression* data type is used to support the relationship between monitor and target program.

**High-level instrumentation** — Information about program execution is available to the monitor from locations in the language runtime system that are coded to report significant events. This obviates the need for, and offers higher performance than target program instrumentation. The runtime system instrumentation is a generalization of an earlier special-purpose monitoring facility oriented around dynamic memory allocation and reclamation [Town89]. It also supercedes Icon and Unicon’s procedure tracing mechanism.

**Events** — The primary language concept added in order to support execution monitoring is an *event*. An event is the smallest unit of execution behavior that is observable by a monitor. In practice, an event is the execution of an instrumentation point in the code (a *sensor*) that is capable of transferring control to the monitor. This definition limits events to those aspects of program behavior that are instrumented in the language runtime system or the program itself. If instrumentation does not exist for an aspect of program behavior of interest, it is often possible to monitor the desired behavior by means of other events. In the present implementation, for example, no instrumentation exists for file input and output. If an EM wishes to monitor I/O behavior, it can monitor function and operator events and act on those functions and operators that relate to input and output. A similar example involving the monitoring of Icon’s built-in string scanning functions is presented in [Jeffery99].

In Unicon, events occur whether they are monitored or not, and each event may or may not be observed by any particular monitor. This definition is useful in a multi-monitor environment, in which EMs are not coupled with the instrumentation and multiple EMs can observe a TP’s execution.

**Event codes and values** — From the monitor’s perspective, an event has two components: an *event code* and an *event value*. The code is generally a one-character
string describing what type of event has taken place. For example, the event code \texttt{C} denotes a procedure call event. Event codes all have associated symbolic constants used in program source code. For example the mnemonic for a procedure call event is \texttt{E_Pcall}. These constants are available to programmers as part of a standard event monitoring library described below.

The event value is an Icon value associated with the event. The nature of an event value depends on the corresponding event code. For example, the event value for a procedure call event is an Icon value designating the procedure being called, the event value for a list creation event is the list that was created, the event value for a source location change event is the new source location, and so forth. Event values can be arbitrary Icon structures with pointer semantics; the EM accesses them just like any other source language value.

**Event reports** — The number of events that occurs during a program execution is extremely large—large enough to create serious performance problems in an interactive system. Most EMs function effectively on a small fraction of the available events; the events that an EM uses are said to be reported to the EM. An event report results in a transfer of control from the TP to the EM. Efficient support for the selection of appropriate events to report and the minimization of the number of event reports are primary concerns.

**Event masks** — A monitor controls the target program by means of this dynamic form of filtering events. An event mask is a set that describes the execution behavior of interest to the monitor. Because event codes are one-letter strings, the \textit{cset} data type is used for event masks. Csets are represented internally by bit vectors, so a cset membership test is very efficient compared to the set data type, whose membership test is a hash table lookup.

Event masking allows the monitor to specify which events are to be reported and to change the specification at runtime. Events that are of no interest to the execution monitor are not reported and do not impose unreasonable execution cost. When a monitor starts or resumes execution of the program being monitored, the monitor selects a subset of possible event codes from which to receive its first report. The program executes until an event occurs with a selected code, at which time the event is reported. After the monitor has finished processing the report, it transfers control back to the program, again specifying an event mask. Dynamic event masking enables the monitor to change the event mask in between event reports.

When an event report transfers control from TP to EM, the two components of the event are supplied in the Icon keywords \texttt{&eventcode} and \texttt{&eventvalue}, respectively. As discussed earlier, these keywords are special global variables that are given their values by the runtime system during an event report, rather than by explicit user
assignment. The monitor then can act upon the event based on its code, display or manipulate its value, etc.

**Standard Execution Monitoring Scenario**

The following scenario presents the relationship between execution monitors and target program in its simplest form. More sophisticated relationships between the monitor and target program, such as running many monitors on a single target program via a monitor coordinator, are described in “Program Monitoring and Visualization” [Jeffery99]. In addition, the expected user and range of program behavior observable using these monitoring facilities are characterized.

**Scenario Definitions**

- **target program (TP)** — The target program is the Unicon program under study, a translated Unicon executable file. Monitoring does not require that the TP be recompiled, nor that the TP’s source code be available, although some monitors make use of program text to present information.

- **execution monitor (EM)** — An execution monitor is a Unicon program that collects and presents information from an execution of a TP.

- **program behavior** — Program behavior denotes the results of executing the TP. Behavior is meant in a general sense that includes program output, execution time, and the precise sequence of actions that take place during execution.

- **user** — In our standard scenario, the user is a human capable of understanding the TP’s execution behavior. The user must know the target language in order to make good use of many EMs or to write a new EM. In general, the user need not necessarily be familiar with the TP’s source code.

Execution monitoring begins with a user who has questions about the behavior of a TP (Figure 9-2). Typical questions relate to correctness or performance, such as “How is the result calculated?” or “What is taking so long?”. Questions may be more general in nature if the user is just trying to understand how a program works.
9.1. MONITOR ARCHITECTURE

Answers to important questions can be found by following the execution at the source language level, but key behavior often depends upon language semantics, implemented by the language runtime system. In Figure 9-3, `iconx.c` denotes the set of files in the Unicon language runtime system. Many monitors can provide useful information about runtime behavior even if the TP’s source code is not available. Figure 9-3 could be elaborated to include dependencies on the platform on which the program is running.

Selecting or Developing Appropriate Monitors

Rather than focusing on one monolithic EM that attempts to accommodate all monitoring tasks, the framework advocates development of a suite of specialized EMs that observe and present particular aspects of a TP’s behavior. The user is responsible for selecting an appropriate EM or set of EMs that address the user’s concerns.

If no available EM can provide the needed information, the user can modify an existing EM or write a new one. This end user development of execution monitors is also useful when an existing EM provides the needed information, but it is obscured by other information; existing EMs can be customized to a particular problem.
Running the Target Program

The user runs the TP, monitored by a selection of EMs (Figure 9-4). General-purpose EMs provide an overall impression of program behavior. Visualization techniques enable the presentation of a large amount of information and abstract away detail.

Obtaining specific information often requires that the user interact with the EMs to control the TP’s execution, either to increase the amount of information presented during specific portions of execution or to pause execution to examine details. In order to provide this interactive control, EMs present execution information as it happens during the TP’s execution, rather than during a postmortem analysis phase.

Framework Characteristics

The preceding scenario requires language support in several areas: controlling a program’s execution, obtaining execution information, presenting large quantities of information, and interacting with the user. To support these tasks, the framework provides synchronous shared address multitasking and an event-driven execution control model. The first two of these features are the focus of this chapter.

Multitasking

In the monitoring execution model, in which an EM is a separate program from the TP, the relationship is almost that of two co-expressions, except that activations of the monitor are implicit occurrences of events within the runtime system, rather than expression results or explicit activations using the @ operator. Event reports are transfers of control to the monitor as well as the primary source of execution information from a TP (Figure 9.5).
9.1. MONITOR ARCHITECTURE

Figure 9-5: EM and TP are separately loaded coroutines

Multitasking has the following benefits for monitoring: the EM and TP are independent programs, the EM has full access to the TP, and the mechanism accommodates multiple EMs. These benefits are described in more detail below.

**Independence**

Because the EM and TP are separate programs, the TP need not be modified or even recompiled in order to be monitored by an EM, and EMs can be used on different target programs. By definition, execution of tasks such as EMs and TPs is synchronous. The TP is not running when an EM is running, and vice versa. This synchronous execution allows EMs and TPs to be independent without introducing the complexities of concurrent programming. In a concurrent context, each thread might have a monitor, but the target thread and its associated monitor are not concurrent — and monitoring concurrent programs is not yet supported by the implementation.

Another degree of EM and TP independence is afforded by separate memory regions; EMs and TPs allocate memory from separate heaps. Memory allocation in the EM does not affect the allocation and garbage collection patterns in the TP. Because Unicon is a type-safe language with runtime type checking and no pointer data types, EMs and TPs cannot corrupt each others’ memory by accident; only code that contains explicit references to another program’s variables and data can modify that program’s behavior. EMs can (and some do) modify TP values in arbitrary ways; the purpose of separate memory regions is to minimize *unintentional* data intrusion.

**Access**

An address space is a mapping from machine addresses to computer memory. Within an address space, access to program variables and data is direct, efficient operations such as single machine instructions. Accessing program variables and data from outside the address space is slow and requires operating system assistance.

The EM and TP reside within the same address space. This allows EMs to treat TP data values in the same way as their own: EMs can access TP structures using regular Unicon operations, compare TP strings with their own, and so forth. Because of the shared
address space, the co-expression switch used to transfer execution between EMs and TPs is a fast, lightweight operation. This is important because monitoring requires an extremely large number of task switches compared to typical multitasking applications.

Multiple Monitors and Monitor Coordinators

Unicon’s dynamic loading capabilities allow simultaneous execution of not just a single EM and a single TP, but potentially many EMs, TPs, and other Icon programs in arbitrary configurations. Although uses for many such configurations can be found, one configuration merits special attention when many specialized EMs are available: the execution of multiple monitors on a single TP (Figure 9-6, left).

The difficulty posed by multiple monitors is not in loading the programs, but in coordinating and transferring control among several EMs and providing each EM with the TP execution information it requires. Since EMs are easier to write if they need not be aware of each other, things are much simpler if EMs run under a monitor coordinator (MC), a special EM that monitors a TP and provides monitoring services to one or more additional EMs (Figure 9-6, right). EMs receiving an MC’s services need not be aware of the presence of an MC any more than a TP need be aware of the presence of an EM.

The virtual monitor interface provided by MCs makes adding a new monitor to the system extremely easy. A new monitor could conceivably be written, compiled, linked, and loaded during a pause in the TP’s execution. In addition, constructing efficient MCs that provide high-level services is another area of research that is supported within the Alamo Icon framework.
Support for Dual Input Streams

An EM typically has two primary input streams: the event stream from the TP, and the input stream from the user (Figure 9-7). Although these two input streams are conceptually independent and may be treated as such, for many EMs this unnecessarily complicates the central loop that obtains event reports from TP—the EM must also check its own window for user activity.

![Figure 9-7: Monitors have two input streams](image)

The runtime system instrumentation includes code that optionally checks for EM input and reports it as an event by the execution monitoring facility, instead of requiring that the EM explicitly check the user input stream. This simplifies EM control flow and improves EM performance.

9.2 Obtaining Events Using evinit

A standard library called evinit provides EMs with a means of obtaining events. Programs wishing to use the standard library include a link declaration such as link evinit. In addition, monitors include a header file named evdefs.icn to obtain the symbolic names of the event codes.

Setting Up an Event Stream

An EM first sets up a source of events; the act of monitoring then consists of a loop that requests and processes events from the TP. Execution monitoring is initialized by the procedure EvInit(x[,input,output,error]). If x is a string, it is used as an icode file name in a call to the Unicon function load(). If x is a list, its first argument is taken as the icode file name and the rest of the list is passed in to the loaded function as the arguments to its main procedure. EvInit() assigns the loaded TP’s co-expression value to EM’s &eventsource keyword. The input, output, and error arguments are files used as the loaded program’s standard files.

EMs generally call the library procedure EvTerm() when they complete, passing it their main window (if they use one) as a parameter. EvTerm() informs the user that execution
has completed and allows the final screen image to be viewed at the user’s leisure, waiting
for the user to press a key or mouse button in the window and then closing it.

The typical EM, and all of the EMs presented as examples in this book, follow the
general outline:

```plaintext
#include "evdefs.icn"
link evinit
procedure main(arguments)
  EvInit(arguments) | stop("can’t initialize monitor")
  # ... initialization code, open the EM window
  # ... event processing loop (described below)
  EvTerm()
end
```

This template is generally omitted from program examples for the sake of brevity.

**EvGet()**

Events are requested by an EM using the function `EvGet(mask)`. `EvGet(mask)` activates
the co-expression value of the keyword `&eventsource` to obtain an event whose code is a
member of the cset `mask`. `mask` defaults to `&cset`, the universal set indicating all events
are to be reported. The TP executes until an event report takes place; the resulting code
and value are assigned to the keywords `&eventcode` and `&eventvalue`. `EvGet()` fails when
execution terminates in TP.

**Event Masks and Value Masks**

`EvGet()` allows a monitor the flexibility to change event masks each time the event source
is activated. Another function that sets event masks is `eventmask()`. `eventmask(C,c)` sets
the event mask of the task owning co-expression C to the cset value given in `c`.

Event masks are the most basic filtering mechanism in Alamo, but there are situations
where they are not specific enough. For example, instead of handling events for all list
operations, you may want events only for specific lists. This situation is supported by the
concept of value masks. A value mask is an Icon set or cset whose members are used to filter
events based on their `&eventvalue`, just as an event mask filters based on the `&eventcode`.
You may specify a different value mask for each event code. Value masks for all event codes
are supplied in a single Icon table value whose keys map event codes to corresponding value
masks. This table is passed as an optional second parameter to `EvGet()` or third parameter
to `eventmask()`. Note that no value mask filtering is performed for event codes that are
not key in the value mask. Note also that value masks persist across calls to `EvGet()`. They
are replaced when a new value mask is supplied, or disabled if a non-table is passed as the
value mask parameter.
9.3. INSTRUMENTATION IN THE ICON INTERPRETER

There is one special case of value masks that receives extra support in Icon: virtual machine instructions. Requesting an event report for the execution of the next virtual machine instruction is performed by calling \texttt{EvGet()} with an event mask containing \texttt{E_Opcode}. VM instructions occur extremely frequently; dozens of them can occur as a result of the execution of a single line of source code. Consequently, performance is severely affected by the selection of all VM instruction events.

However, a particular instruction or small set of instructions may be of interest to a monitor. In that case, the EM need not receive reports for all instructions. The function \texttt{opmask(C, c)} allows EM to select a subset of virtual machine instructions given by \texttt{c} in \texttt{C}'s task. Subsequent calls to \texttt{EvGet()} in which \texttt{E_Opcode} is selected reports events only for the VM instructions designated by \texttt{c}.

The event values for \texttt{E_Opcode} are small non-negative integers. They fall in a limited range ($< 256$), which is what allows a cset representation for them. Symbolic names for individual virtual machine instructions are defined in the include file \texttt{opdefs.icn}. \texttt{opmask(C,c)} is equivalent to:

\begin{verbatim}
  t := table()
  t[E_Opcode] := c
  eventmask(C, , t)
\end{verbatim}

9.3 Instrumentation in the Icon Interpreter

This section describes the instrumentation used by Unicon to produce events at various points in the runtime system. Significant points in interpreter execution where transfer of control might be warranted are explicitly coded into the runtime system with tests that result in transfer of control to an EM when they succeed. When execution reaches one of these points, an event occurs. Events affect the execution time of the TP; execution is either slowed by a test and branch instruction (if the event is not of interest to the EM), or stopped while the event is reported to the EM and it processes information. Minimizing the slowdown incurred due to the presence of monitoring instrumentation has been a focus of the implementation.

There are several major classes of events that have been instrumented in the Unicon interpreter. Most of these events correspond to explicit elements within the source code; others designate actions performed implicitly by the runtime system that the programmer may be unaware of. A third class of event that has been instrumented supports user interaction with the EM rather than TP behavior.

Explicit Source-Related Events

The events that relate behavior observable from the source code are:
Program location changes — Source code locations are reported in terms of line numbers and columns.

Procedure activity — There are events for procedure calls, returns, failures, suspensions, and resumptions. In addition to these explicit forms of procedure activity, events occur for implicit removals of procedure frames.

Built-in functions and operations — Events that correspond to Icon built-ins describe many areas of behavior from numeric and string operations to structure accesses and assignments. Like procedures, events are produced for function and operator calls, returns, suspensions, resumptions, and removals.

String scanning activity — Icon’s pattern matching operations include scanning environment creation, entry, change in position, and exit. To obtain a complete picture of string scanning, monitors must observe these events along with the built-in functions related to string scanning.

Implicit Runtime System Events

Events that depict important program behavior observed within the runtime system include:

Memory allocations — Memory is allocated from the string and block regions in the heap. Allocation events include size and type information. This instrumentation is based on earlier instrumentation added to Icon for a memory monitoring and visualization system [Town89].

Garbage collections — The storage region being collected (Icon has separate regions for strings and data structures), the memory layout after compaction, and the completion of garbage collection are reported by several events.

Type conversions — In Icon, automatic conversions are performed on parameters to functions and operators. Information is available for conversions attempted, failed, succeeded, and found to be unnecessary.

Virtual machine instructions — Icon’s semantics may be defined by a sequence of instructions executed by the Icon virtual machine [Gris86]. The program can receive events for all virtual machine instructions, or an arbitrary subset.

Clock ticks — The passage of CPU time is indicated by a clock tick.

Most EMs, except completely passive visualizations and profiling tools, provide the user with some degree of control over the monitoring activity and must take user interaction into
account. For example, the amount of detail or the rate at which the monitor information is updated may be variables under user control. Since an EM’s user input occurs only as often as the user presses keys or moves the mouse, user interaction is typically far less frequent than events in TP. Even if no user input occurs, polling for user input may impose a significant overhead on the EM because it adds code to the central event processing loop.

In order to avoid this overhead, the event monitoring instrumentation includes support for reporting user activity in the EM window as part of the TP’s event stream. Monitor interaction events are requested by the event code E\_MXevent. An example of the use of monitor interaction events is presented further in this chapter in the section entitled “Handling User Input”. A complete list of event codes is presented in Appendix ?? in order to indicate the extent of the instrumentation.

### 9.4 Artificial Events

As described above, the Unicon co-expression model allows interprogram communication via explicit co-expression activation or implicit event reporting within the runtime system. Artificial events are events produced by explicit Icon code; they can be viewed at the language level as co-expression activations that follow the same protocol as implicit events, assigning to the keyword variables &eventcode and &eventvalue in the co-expression being activated.

There are two general categories of artificial events, virtual events meant to be indistinguishable from implicit events and pseudo events that convey control messages to an EM. Virtual events are generally used either to produce event reports from manually instrumented locations in the source program, to simulate event reports, or to pass on a real event from the primary EM that received it to one or more secondary EMs. Pseudo events, on the other hand, are used for more general inter-tool communications during the course of monitoring, independent of the TP’s execution behavior.

#### Virtual Events Using event()

The function event(code, value, recipient) sends a virtual event report to the co-expression recipient, which defaults to the &main co-expression in the parent of the current task, the same destination to which implicit events are reported.

There are times when a primary EM wants to pass on its events to a secondary EM. An example would be an event transducer that sits in between the EM and TP, and uses its own logic to determine which events are reported to EM with more precision than is provided by the masking mechanism. A transducer might just as easily report extra events with additional information it computes, in addition to those received from TP. A more substantial application of virtual events is a monitor coordinator, an EM that coordinates and produces events for other monitors. Such a tool is presented in [Jeffery99]
Pseudo Events for Tool Communication

EMs generally have an event processing loop as their central control flow mechanism. The logical way to communicate with such a tool is to send it an event. In order to distinguish a message from a regular event report, the event code must be distinguishable. In the monitoring framework, this is achieved simply by using an event code other than a one-letter string, such as an integer. Since not all EMs handle such events, they are not delivered to an EM unless it passes a non-null second argument (the “value mask argument”) to `EvGet()`, such as `EvGet(mask, 1)`.

The framework defines a minimal set of standard pseudo events, which well-behaved EMs should handle correctly [Jeffery99]. Beyond this minimal set, pseudo events allow the execution monitor writer to explore communication between EMs as another facility to ease programming tasks within the monitoring framework.

9.5 Monitoring Techniques

Monitors generally follow a common outline and use a common set of facilities, which are described below.

Anatomy of an Execution Monitor

The execution monitoring interface uses a form of event driven programming: the central control flow of EM is a loop that executes the TP for some amount of time and then returns control to EM with information in the form of an event report. The central loop of an EM typically looks like:

```pseudocode
while EvGet(eventmask) do
    case &eventcode of {
        # a case clause for each code in the event mask
    }
```

Event-driven programming is more commonly found in programs that employ a graphical user interface, where user activity dominates control flow. Because monitoring employs a programming paradigm that has been heavily studied, many coding techniques developed for graphical user interface programming, such as the use of callbacks [Clark85], are applicable to monitors. Several of the example EMs in the IPL use a callback model to take advantage of a higher-level monitoring abstraction available by means of a library procedure.

Handling User Input

An EM that handles user input could do so by polling the window system after each event in the main loop:
9.5. MONITORING TECHNIQUES

while EvGet(eventmask) do {
    case &eventcode of {
        # a case clause for each code in the event mask
    }
    # poll the window system for user input
}

If the events being requested from the TP are relatively infrequent, this causes no great problem. However, the more frequent the event reports are, the more overhead is incurred by this approach relative to the execution in TP. In typical EMs, polling for user events may slow execution from an imperceptible amount to as much as 15 percent. Relative frequency for different types of events varies wildly; it is discussed in [Jeffery99].

Since the slowdown is a function of the frequency of the event reports and not just the cost of the polling operation itself, techniques such as maintaining a counter and polling once every \( n \) event reports still impose a significant overhead. In addition, such techniques reduce the responsiveness of the tool to user input and therefore reduce the user’s control over execution.

Monitor interaction events, presented earlier in this chapter, address this performance issue by allowing user input to be supplied via the standard event stream produced by \texttt{EvGet()}. Since the \texttt{E\_MXevent} event occurs far less frequently than other events, it makes sense to place it last in the case expression that is used to select actions based on the event code. Using this feature, the main loop becomes:

while EvGet() do
    case &eventcode of {
        # other cases update image to reflect the event
        E\_MXevent: {
            # process user event
        }
    }

\texttt{EvGet()} reports pending user activity immediately when it is available; the control over execution it provides is comparable to polling for user input on each event.

After each event report, EMs can use Unicon’s intertask data access functions to query TP for additional information, such as the values of program variables and keywords. The access functions can be used in several ways, such as

- applying a predicate to each event report to make monitoring more specific,
- \textit{sampling} execution behavior not reported by events by polling the TP for information unrelated to the event reports [Ogle90], or
- presenting detailed information to the user, such as the contents of variables.
9.6 Some Useful Library Procedures

As mentioned, several library procedures are useful in EMs. The following library procedures that are included in the `evinit` library.

Location decoding and encoding procedures are useful in processing location change event values, but they are also useful in other monitors in which two-dimensional screen coordinates must be manipulated. Besides program text line and columns, the technique can variously be applied to individual pixels, to screen line and columns, or to screen grid locations in other application-specific units.

In addition, various EMs use utility procedures. Figure 9-8 lists some library procedures that are recommended for use in monitors.

<table>
<thead>
<tr>
<th>procedure</th>
<th>returns or computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>evnames(s)</td>
<td>converts event codes to text descriptions and vice versa</td>
</tr>
<tr>
<td>evsyms()</td>
<td>two-way table mapping event codes to their names</td>
</tr>
<tr>
<td>typebind(w,c)</td>
<td>table mapping codes to color coded Clones for w</td>
</tr>
<tr>
<td>opnames()</td>
<td>table mapping VM instructions to their names</td>
</tr>
<tr>
<td>location()</td>
<td>encodes a two dimensional location in an integer</td>
</tr>
<tr>
<td>vertical()</td>
<td>y/line/row component of a location</td>
</tr>
<tr>
<td>horizontal()</td>
<td>x/column component of a location</td>
</tr>
<tr>
<td>prog_len()</td>
<td>number of lines in the source code for TP</td>
</tr>
<tr>
<td>procedure_name()</td>
<td>name of a procedure</td>
</tr>
<tr>
<td>WColumns()</td>
<td>window width in text columns</td>
</tr>
<tr>
<td>WHeight()</td>
<td>window height in pixels</td>
</tr>
<tr>
<td>WRows()</td>
<td>window height in text rows</td>
</tr>
<tr>
<td>WWidth()</td>
<td>window width in pixels</td>
</tr>
</tbody>
</table>

Figure 9-8: Additional library procedures for monitors.

9.7 Conclusions

Unicon includes facilities to exploit instrumentation available within its runtime system. Writing a monitor consists of writing an ordinary application. The key concepts introduced for Unicon’s event monitoring facilities are events, event reports, event codes and values, and event masks. Monitors also make use of a standard monitoring library and the graphics facilities.
Part II

Object-oriented Software Development
Chapter 10

Objects and Classes

Object-oriented programming means different things to different people. In Unicon, object-oriented programming starts with encapsulation, inheritance, and polymorphism. These ideas are found in most object-oriented languages as well as many that are not object-oriented. This and following chapters present these ideas and illustrate their use in design diagrams and actual code. Diagrams and code are alternative notations by which programmers share their knowledge. This chapter explores the essence of object-orientation and gives you the concepts needed before you delve into diagrams and code examples. In this chapter you will learn:

- How different programming languages support objects in different ways
- To simplify programs by encapsulating data and code
- The relationship between objects and program design
- Draw diagrams that show class names, attributes, and methods
- Write corresponding code for classes and their methods
- To create instances of classes and invoke methods on those objects

10.1 Objects in Programming Languages

Object-oriented programming can be done in any language, but some languages make it easier than others. Support for objects should not entail strange syntax or programs that look funny in a heterogeneous desktop-computing environment. Smalltalk has these problems. C++ avoids these programs, but its low-level machine-orientation is less than ideal as an algorithmic notation usable by non-experts. Java offers a simple object model and familiar syntax. The advantages Unicon has over Java are fundamentally higher-level built-in types, operations, and control structures.

Many object-oriented languages require that \textit{everything} be done in terms of objects, even when objects are not appropriate. Unicon provides objects as just another tool to aid
in the writing of programs, especially large ones. Icon already provides a powerful notation for expressing a general class of algorithms. The purpose of object-orientation is to enhance that notation, not to get in the way of it.

Icon does not support user-defined objects, although its built-in types have nice object-like encapsulation and polymorphism properties. Unicon’s object-oriented facilities descend from a package for Icon called Idol. In Idol, a preprocessor implemented objects with no support from the underlying Icon runtime system. In contrast, Unicon has support for objects built-in to the language. This simplifies the notation and improves the performance of object-related computations.

Object-orientation adds several general concepts into procedure-based programming. The single overriding reason for object-oriented programming is to reduce complexity in large programs. Simple programs can be written easily in any language. Somewhere between the 1,000-line mark and the 10,000-line mark most programmers can no longer keep track of their entire program at once. By using a very high-level programming language, fewer lines of code are required; a programmer can write perhaps ten times as large a program and still be able to keep track of things.

As programmers write larger and larger programs, the benefit provided by very high-level languages does not keep up with program complexity. This obstacle has been labeled the “software crisis”, and object-oriented programming is one way to address this crisis. In short, the goals of object-oriented programming are to reduce the complexity of coding required to write very large programs and to allow code to be understood independently of the context of the surrounding program. The techniques employed to achieve these goals are discussed below.

A second reason to consider object-oriented programming is that the paradigm fits certain problem domains especially well, such as simulation, and graphical user interfaces. The first well-known object-oriented language, Simula67, certainly had the domain of simulation in mind. The second pioneering object-oriented language, Smalltalk, popularized fundamental aspects of bitmapped graphical user interfaces that are nearly universal today. Three decades of experience with object-oriented techniques has led many practitioners to conclude that the concepts presented below are very general and widely applicable, but not all problems fit the object-oriented mold. Unicon advocates the use of objects, but this is a suggestion, not a rule.

**Encapsulation**

The primary concept advocated by object-oriented programming is the principle of encapsulation. Encapsulation is the isolation, in the source code that a programmer writes, of a data representation and the code that manipulates the data representation. In some sense, encapsulation is an assertion that no other routines in the program have *side-effects* with respect to the data structure in question. It is easier to reason about encapsulated data
because all of the source code that could affect that data is immediately present with its definition.

Encapsulation does for data structures what the procedure does for algorithms: it draws a line of demarcation in the program source code. Code outside this boundary is irrelevant to the code that is inside, and vice versa. Communication across the boundary occurs through a public interface. An encapsulated data structure is called an object. Just as a set of named variables called parameters is the interface between a procedure and the code that uses it, a set of named procedures called methods comprises the interface between an object and the code that uses it.

This textual definition of encapsulation as a property of program source code accounts for the fact that good programmers can write encapsulated data structures in any language. The problem is not capability, but verification. To verify encapsulation some languages require programmers to specify the visibility of every piece of information in each data structure as public or private. There are even multiple forms of privacy (semi-private?). Unicon instead stresses simplicity.

Inheritance

In large programs, the same or nearly the same data structures are used over and over again for myriad different purposes. Similarly, variations on the same algorithms are employed by structure after structure. To minimize redundancy, techniques are needed to support code sharing for both data structures and algorithms. Code is shared by related data structures through a programming concept called inheritance.

The basic premise of inheritance is simple: when writing code for a data structure similar to a structure that is already written, specify the new structure by giving the differences between it and the old structure, instead of copying and modifying the old structure’s code. There are times when the inheritance mechanism is not useful, such as if the two data structures are more different than they are similar, or if they are simple enough that inheritance would only confuse things, for example.

Inheritance addresses multiple programming problems found at different conceptual levels. The most obvious software engineering problem it solves might be termed enhancement. During the development of a program, its data structures may require extension via new state variables or new operations or both; inheritance is especially useful when both the original structure and the extension are used by the application. Inheritance also supports simplification, or the reduction of a data structure’s state variables or operations. Simplification is analogous to argument culling, an idea from lambda calculus (don’t worry if this sounds like Greek to you), in that it describes a logical relation between structures. In general, inheritance may be used in source code to describe any sort of relational hyponymy, or special casing. In Unicon the collection of all inheritance relations defines a directed (not necessarily acyclic) graph.
Polymorphism

From the perspective of the writer of related data structures, inheritance provides a convenient method for code sharing, but what about the code that uses objects? Since objects are encapsulated, that code is not dependent upon the internals of the object at all, and it makes no difference to the client code whether the object in question belongs to the original class or the inheriting class.

We can make a stronger statement. Due to encapsulation, different executions of code that uses objects to implement an algorithm may operate on objects that are not related by inheritance at all. Such code can utilize any objects that implement the operations that the code invokes. This facility is called polymorphism, and such algorithms are called generic. This feature is found in many non-object-oriented languages; in object-oriented languages it is a natural extension of encapsulation.

10.2 Objects in Program Design

Another fundamental way to think about objects is from the point of view of software design. During program design, objects are used to model the problem domain. The different kinds of objects and relationships between objects capture fundamental information that organizes the rest of the program’s design and implementation. Program design includes several other fundamental tasks such as the design of the user interface, or interactions with external systems across a network. Additional kinds of modeling are used for these tasks, but they all revolve around the object model.

For small, simple, or well-understood software projects, a prose description may be all the documentation that is needed. The Unified Modeling Language (UML) is a notation for building software models of larger software systems for which a prose description alone would be inadequate. It was invented by Grady Booch, Ivar Jacobson, and James Rumbaugh. In UML, software models document the purpose and function of a software system. The advantage of a model is that it conveys information that is both more precise and more readily understood than a prose description. UML is used during multiple phases of the software lifecycle. UML defines several kinds of diagrams, of which we will only consider four in this book.

- **Use case diagrams** show the organization of the application around the specific tasks accomplished by different users.

- **Class diagrams** show much of the static structure of the application data.

- **Statechart diagrams** model dynamic behavior of systems that respond to external events, including user input.

- **Collaboration diagrams** model interactions between multiple objects
These diagrams describe key aspects of many categories of software applications. The reader should consult the UML Notation Guide and Semantics documents for a complete description of UML. A good introduction is given in *UML Toolkit*, by Hans-Erik Eriksson and Magnus Penker (1998).

A typical application of UML to a software development project uses these diagrams in sequence. You start by constructing use case diagrams and detailed descriptions of the different kinds of users and tasks performed using the system. Then develop class diagrams that capture the relationships between different kinds of objects in the system. Finally, construct statechart and collaboration diagrams as needed to describe the sequences of events that can occur and the corresponding operations performed by various objects in response to those events.

Use case and statechart diagrams are important, but their purpose is to elaborate on an object model described in class diagrams. For this reason, class diagrams are presented first, along with the corresponding programming concepts. Use case diagrams, statecharts, and collaboration diagrams are discussed in Chapter 12.

### 10.3 Classes and Class Diagrams

Classes are user-defined data types that model the information and behavior of elements in the application domain. In Unicon they are records with associated procedures, called methods. Instances of these special record types are called objects. But the language constructs originated from a need to model application domain concepts, so it is appropriate to introduce them from that perspective.

Modeling a software system begins with identifying things that are in the system and specifying how they are related. A class diagram shows a static view of relationships between the kinds of elements that occur in the problem domain. A class diagram is a data-centric, object-centric model. In contrast, a user-centric view is provided by use cases. Class diagrams have several basic components.

Classes are represented by rectangles. A *class* denotes a concept of the application domain that has state information (depicted by named *attributes*) and/or behavior (depicted by named operations, or *methods*) significant enough to be reflected in the model. Inside the rectangle, lines separate the class name and areas for attributes and operations. Figure 10-1 shows an example class.
Classes in an object model are implemented using programming language classes, which are described in the next section. The degree of separation between the notion of a class in the model and in the implementation depends on the programming language. In the case of Unicon, the separation is minimal, because built-in types such as lists and tables take care of almost all data structures other than those introduced specifically to model application domain elements. In the case of C++ or Java, many additional implementation artifacts typically have to be represented by classes.

The same class can appear in many class diagrams to capture all of its relationships with other classes. Different diagrams may show different levels of detail, or different aspects (projections) of the class relevant to the portion of the model that they depict. In the course of modeling it is normal to start with few details and add them gradually through several iterations of the development process. Several kinds of details may be added within a class. Such details include:

- The visibility of attributes and operations. A plus sign (+) before the attribute name indicates that the attribute is public and may be referenced in code external to the class. A minus sign (-) before the attribute name indicates that the attribute is private and may not be referenced in code outside the class.

- Types, initial values, and properties of attributes

- Static properties of the class that will not be relevant at run-time

Attribute names may be suffixed with a colon and a type, an equal sign and a value, and a set of properties in curly braces. Figure 10-2 shows two very different levels of detail for the same class. Each level of detail is appropriate in different contexts.
10.4 Declaring Classes

In Unicon program code, the syntax of a class is:

```unicon
class foo(attribute1, attribute2, attribute3, ...)
    # procedures (methods) to access class foo objects

    # code to initialize class foo objects
end
```

The procedures that manipulate class objects are called methods. Methods define a class’ interface from the rest of the program. The syntax of a method like a procedure:

```unicon
method bar(param1, param2, param3, ...)
    # Unicon code that may access fields of a class foo object
end
```

Execution of a class method is always associated with a given object of that class. The method has access to an implicit variable called `self` that is a record containing fields whose names are the attributes given in the class declaration. Fields from the `self` variable are directly accessible by name. In addition to methods, classes may also contain regular procedure, global, and record declarations; such declarations have the standard semantics and exist in the global name space.
10.5 Object Instances

Like records, instances of a class type are created with a constructor function whose name is that of the class. Instances of a class are called objects, and behave similar to records. The fields of an instance generally correspond directly to the class attributes. Fields may be initialized explicitly in the constructor in exactly the same way as for records. For example, after defining a class `foo(x, y)` one may write:

```plaintext
procedure main()
    f := foo(1, 2)
end
```

In this case `x` would have the value 1, and `y` would have the value 2, the same as for a record type. The fields of an object do not have to be initialized by a parameter passed to the class constructor. Many constructors initialize objects’ fields to some standard value. In this case, the class declaration includes an `initially` section after its methods are defined and before its end. An `initially` section is just a special method that is invoked automatically by the system when it creates each instance of the class.

An `initially` section begins with the word `initially` and an optional parameter list, followed by lines of code that are executed when an object of that class is constructed. These lines typically assign values to one or more of the attributes of the object being created.

For example, suppose you want an enhanced table type that permits sequential access to elements in the order they are inserted into the table. You can implement this using a combination of a list and a table, both of which would be initialized to the appropriate empty structure:

```plaintext
class taque(L, T) # pronounced "taco"
    # methods to manipulate taques,
    # e.g. insert, index, foreach...
    initially
        L := []
        T := table()
end
```

In such a case you can create objects without including arguments to the constructor:

```plaintext
procedure main()
    mytaque := taque()
end
```

Although the default behavior of classes is the same as records, and constructor arguments normally assign each of the fields in the class in order, there are some important rules that override classes’ record-like constructor behavior.
• With no `initially` section, a class constructor behaves exactly like a record constructor. Class fields are assigned in order from parameters and missing arguments default to the null value.

• With an `initially(...)` that has a parenthesized list of zero or more formal parameter names, the constructor parameters are used to initialize the formal parameters of the `initially()` method. In this case, the constructor does not assign class fields from parameters implicitly. Instead the initially section may initialize fields as it sees fit, including initializing them using values from the named parameters of method `initially()` if it so chooses.

• An `initially` section without a parenthesized formal parameter list behaves somewhere in between the above two cases. Normal record-like assignment of parameters to fields is performed. Missing arguments start with a null value. However, the `initially` section may assign those fields without the caller having passed values into the constructor.

The initially section with no parameter list makes it possible to write classes with some fields that are initialized explicitly by the constructor parameters, while other fields are initialized implicitly by code in the `initially` section. In this case you should declare the automatically initialized fields after those initialized by parameters in the constructor. The parameters are assigned to fields in the constructor exactly in the order they appear in the class declaration.

This default semantics for constructor parameters is awkward in some cases, so there is an alternative. When an initially section includes a parameter list, no implicit initialization of objects' fields is performed. This frees the constructor from having the same number and order of parameters as the declared class fields. In the following example, class `C` is constructed with only a single parameter even though it has three fields. The actual parameter "Greenwich Village" is bound to `initially` formal parameter `x`. The third field in the class (`c`) is initialized from the constructor parameter `x`, overriding the default behavior of initializing fields in the declared order. This capability becomes important in large classes with many fields.

```plaintext
class C(a, b, c)
    initially(x)
        a := ["vital", "urgent", "gentrified"]
        b := promptedread("tell me about " || x)
        c := x
    end
... 
procedure main()
    v := C("Greenwich Village")
end
```
10.6 Object Invocation

Once you have created an object with a class constructor, you manipulate the object by invoking its class methods. Since objects are both procedures and data, object invocation is a combination of a procedure call and a record access. The syntax is

\[
\text{object . methodname ( arguments )}
\]

If an object’s class is known, object methods can also be called using a normal procedure call. This allows object oriented Unicon code to be called from Icon. Called as a procedure, the name of a method is prefixed with the class name and an underscore character. The object itself is always the first parameter passed to a method. In the absence of inheritance (discussed in the next chapter) if \( x \) is an object of class \( C \), \( x\text{.method}(\text{arguments}) \) is equivalent to \( C\_\text{method}(x, \text{arguments}) \).

Although object methods can be called using procedure calls, the field operator has the advantage that it handles inheritance and polymorphism correctly, allowing algorithms to be coded generically using polymorphic operations. Generic algorithms use any objects whose class provides the set of methods used in the algorithm. Generic code is less likely to require change when you later enhance the program, for example adding new subclasses that inherit from existing ones. In addition, if class names are long, the field syntax is considerably shorter than writing out the class name for the invocation. Using the taque example:

```unicon
procedure main()
  mytaque := taque()
  mytaque.insert("greetings", "hello")
  mytaque.insert(123)
  every write(mytaque.foreach())
  if mytaque.index("hello") then write("", world)
end
```

For object-oriented purists, using the field operator to invoke an object’s methods in this manner is the only way to access an object. In Unicon, visibility issues such as “public” and “private” are addressed in an application’s design and documentation. A good starting point is to consider all fields “private” and all methods “public”. Nevertheless, an object is just a kind of record, complete with record-style field access.

Direct external access to an object’s data fields using the usual field operator is not good practice, since it violates the principle of encapsulation. Within a class method, on the other hand, access to an object’s attributes is expected. The implicit object named `self` is used under the covers, but attributes and methods are referenced by name, like other variables. The taque insert method is thus:
10.7. COMPARING RECORDS AND CLASSES

```unicon
method insert(x, key)
    /key := x
    put(L, x)
    T[key] := x
end
```

The self object allows field access just like a record, as well as method invocation like any other object. Using the self variable explicitly is rare.

10.7 Comparing Records and Classes

The concepts of classes and objects are found in many programming languages. The following example illustrates Unicon’s object model and provides an initial impression of these concepts’ value. To motivate Unicon’s OOP constructs, our example contrasts conventional Icon code with object-oriented code that implements the same behavior.

**Before objects**

Suppose you are writing some text-processing application such as a text editor. Such applications need to be able to process structures holding the contents of various text files. You might begin with a simple structure like the following:

```icon
record buffer(filename, text, index)
```

where filename is a string, text is a list of strings corresponding to lines in the file, and index marks the current line at which the buffer is being processed. Icon record declarations are global; in principle, if the above declaration needs to be changed, the entire program must be rechecked. A devotee of structured programming would write procedures to read the buffer in from a file; write it out to a file; examine, insert and delete individual lines; and so on. These procedures, along with the record declaration given above, can be placed in their own source file (buffer.icn) and understood independently of the program(s) in which they are used. Here is one such procedure:

```icon
# read a buffer in from a file
procedure read_buffer(b)
    f := open(b.filename) | fail
    b.text := [ ]
    b.index := 1
    every put(b.text, !f)
    close(f)
    return
end
```
There is nothing wrong with this example; in fact its similarity to the object-oriented example that follows demonstrates that a good, modular design is the primary effect encouraged by object-oriented programming. Using a separate source file to contain a record type and those procedures that operate on the type allows an Icon programmer to maintain a voluntary encapsulation of that type.

After objects

Here is part of the same buffer abstraction coded in Unicon. The purpose here is to facilitate a direct comparison with the preceding record-based example. The example lays the groundwork for a further object-oriented illustration in the following chapter. Classes are record types with associated code. Methods are procedures that are always called in reference to a particular object (a class instance).

```unicon
class buffer(filename, text, index)
    # read a buffer in from a file
    method read()
        f := open(filename) | fail
        text := [ ]
        index := 1
        every put(text, !f)
        close(f)
        return
    end
    # ...additional buffer operations, including method erase()
initially
    if \filename then read()
end
```

This example does not illustrate the full object-oriented style, but it is a start. The object-oriented version offers encapsulation and polymorphism. A separate name space for each class’s methods allows shorter names. The same method name, such as `read()`, can be used in each class that implements a given operation. This notation is more concise than is possible with procedures, and it enables an algorithm to work on objects of any class that implements the operations required by that algorithm.

Consider the initialization of a new buffer. Constructors allow the initialization of fields to values other than `&null`. In the example, the `read()` method is invoked if a filename is supplied when the object is created. This can be simulated using records by calling a procedure after the record is created; the value of the constructor is that it is automatic. The programmer is freed from the responsibility of remembering to call this code everywhere objects are created in the client program(s). This tighter coupling of memory allocation and its corresponding initialization removes one more source of program errors, especially on multi-programmer projects.
The preceding two paragraphs share a common theme: the net effect is that each piece of data is made responsible for its own behavior in the system. Although this example dealt with simple line-oriented text files, the same methodology applies to more abstract entities such as the components of a compiler’s grammar.

The example illustrates an important scoping issue. Within class buffer, method `read()` makes the regular built-in function `read()` inaccessible! Beware of such conflicts. It would be easy to capitalize the method name to eliminate the problem. If renaming the method is not an option, as a last resort you could get a reference to the built-in function `read()`, even within method `read()`, by calling `proc("read", 0)`. The function `proc()` converts a string to a procedure; supplying a second parameter of 0 tells it to skip scoping rules and look for a built-in function by that name.

### 10.8 Summary

Classes are global declarations that define a record data type and a set of procedures (methods) that operate on that type. Class instances, called objects, are normally manipulated solely by calling the class’ methods; such object privacy is a matter of design, documentation, and convention. All methods execute within an object of interest, whose fields and methods are visible without the record dot notation, in a class scope that is introduced in between the local and global scopes.
Chapter 11

Inheritance and Associations

Relationships between classes are depicted in UML class diagrams by lines drawn between two or more class rectangles. One of the most important relationships between classes describes the situation when one class is an extension or minor variation of another class: this is called generalization, or inheritance. Most other relationships between classes are really relationships between those classes’ instances at run-time; these relationships are called associations. This chapter starts with inheritance, and then describes a variety of associations. In this chapter you will learn how to:

- Define a new class in terms of its differences from an existing class
- Compose aggregate classes from component parts.
- Specify new kinds of associations
- Supply details about the roles and number of objects in an association
- Use structure types to implement associations

11.1 Inheritance

In many cases, several classes of objects are very similar. In particular, many classes arise as enhancements of classes that have already been defined. Enhancements might consist of added fields, added methods, or both. In other cases a class is just a special case of another class. For example, if you have a class \( \text{fraction}(\text{numerator, denominator}) \), you could define class \( \text{inverses}(\text{denominator}) \) whose behavior is identical to that of a fraction, but whose numerator is always 1.

Both of these ideas are realized with the concept of inheritance. When the definition of a class is best expressed in terms of the definition of another class or classes, we call that class a subclass. The class or classes from which a subclass obtains its definition are called superclasses. The logical relation between the subclass and superclass is called hyponymy. It means an object of the subclass can be manipulated just as if it were an object of one
of its defining classes. In practical terms it means that similar objects can share the code that manipulates their fields.

Inheritance appears in a class diagram as a line between classes with an arrow at one end. The arrow points to the superclass, the source of behavior inherited by the other class. Consider Figure 11-1, in which an offer of a salaried appointment is defined as one kind of job offer. The attributes (\texttt{salary, title}) and methods (\texttt{Accept(), Reject()}) of class JobOffer are inherited by class SalariedAppointment, and do not need to be repeated there. A new attribute (term) is added in \texttt{SalariedAppointment} that is not in \texttt{JobOffer}.

![Figure 11-1: A Salaried Appointment is a subclass of a Job Offer](image)

The syntax of a subclass is

```
class classname superclasses (attributes) 
    methods
    initially _section
end
```

Where \texttt{superclasses} is an optional list of class names separated by colons, \texttt{attributes} is an optional list of variable names separated by commas, \texttt{methods} is an optional list of declarations for the class methods, and the \texttt{initially _section} is optional initialization code for the class constructor. For example

```
class SalariedAppointment : JobOffer (term)
    method SetTerm(t : interval)
        term := t
    end
    initially
        /term := "unknown term"
end
```

As you can see, a subclass declaration is identical to a regular class, with the addition of one or more superclass names, separated by colons. The meaning of this declaration is the subject of the next section.
11.1. INHERITANCE

Inheritance semantics

There are times when a new class might best be described as a combination of two or more classes. Unicon classes may have more than one superclass, separated by colons in the class declaration. This is called multiple inheritance.

Subclasses define a record type consisting of the field names of the subclass itself and all its superclasses. The subclass has associated methods consisting of those in its own body, those in the first superclass that were not defined in the subclass, those in the second superclass not defined in the subclass or the first superclass, and so on. In ordinary single-inheritance, this adding of fields and methods follows a linear bottom-up visit of each superclass, followed in turn by its parent superclass.

When a class has two or more superclasses, the search generalizes from a linear sequence to an arbitrary tree, directed acyclic graph, or full graph traversal. Multiple inheritance adds fields and methods in an order defined by a depth-first traversal of the parent edges of the superclass graph. Think of the second and following superclasses in the multiple inheritance case as adding methods and fields only if the single-inheritance case (following the first superclass and all its parents) has not already added a field or method of the same name.

Warning

Care should be taken employing multiple inheritance if the two parent classes have any fields or methods of the same name!

Fields are initialized by parameters to the constructor or by the class initially section, which is a method and is inherited in the normal way. It is common for a subclass initially section to call their superclasses’ initially sections, for example:

```unicon
class sub : A : B(x)
  initially
    x := 0
    A.initially()
    B.initially()
end
```

It is also common to have some attributes initialized by parameters, and assign others in the initially section. For example, to define a class inverse (for numbers of the form 1 / n) in terms of a class fraction(numerator, denominator) one can write:

```unicon
class inverse : fraction (denominator)
  initially
    numerator := 1
end
```

Objects of class inverse can be manipulated using all the methods defined in class fraction; the code is actually shared by both classes at runtime.
Viewing inheritance as the addition of superclass elements not found in the subclass is the opposite of the more traditional object-oriented view that a subclass is an instance of the superclass as augmented in the subclass definition. Unicon’s viewpoint adds quite a bit of leverage, such as the ability to define classes that are subclasses of each other. This feature is described further below.

**Invoking superclass operations**

When a subclass defines a method of the same name as a method defined in the superclass, invocations on subclass objects execute the subclass’ version of the method. This can be overridden by explicitly including the superclass name in the invocation:

```
object$superclass.method(parameters)
```

This facility allows the subclass method to do any additional work required for added fields before or after calling an appropriate superclass method to achieve inherited behavior. The result is frequently a chain of inherited method invocations.

Since initially sections are methods, they can invoke superclass operations including superclass initially sections. This allows a chain of initially sections to be specified to execute in either subclass-first or superclass-first order, or some mixture of the two.

**Inheritance examples**

Several inheritance examples follow from the buffer example from Chapter 10. Suppose the program does more than edit text, it has a word-associative dictionary, bibliography, spell-checker, and thesaurus. These features can be implemented using the table type. The contents vary, but they all use string keyword lookup. As external data, the databases can be stored in text files, one entry per line, with the keyword at the beginning. The format of the rest of the line varies from database to database.

Figure 11-2 shows a class diagram with subclasses derived from buffer. A class buftable refines the buffer class, adding support for random access. Other classes are defined as subclasses of buftable. The implementation of these classes is given below.
Although all these types of data are different, the code used to read the data files can be shared, as well as the initial construction of the tables. In fact, since we are storing our data one entry per line in text files, we can use the code already written for buffers to do the file I/O itself.

```lisp
class buftable : buffer()
method read()
    self.buffer.read()
    tmp := table()
    every line := !text do
        line ? { tmp[tab(many(&letters))] := line | fail }
        text := tmp
    end
    return
end
method index(s)
    return text[s]
end
end
```

This concise example shows how little must be written to achieve data structures with vastly different behavioral characteristics, by building on code that is already written. The superclass `read()` operation is one important step of the subclass `read()` operation. This technique is common enough to have a name: it is called *method combination* in the literature. It allows you to view the subclass as a transformation of the superclass. The `buftable` class is given in its entirety, but our code sharing example is not complete: what about the data structures required to support the databases themselves? They are
all variants of the \texttt{buftable} class, and a set of possible implementations follow. Note that the formats presented are designed to illustrate code sharing; clearly, an actual application might make different choices.

**Bibliographies** Bibliographies might consist of a keyword followed by an uninterpreted string of information. This imposes no additional structure on the data beyond that imposed by the \texttt{buftable} class. An example keyword would be Jeffery99.

\begin{verbatim}
class bibliography : buftable()
end
\end{verbatim}

**Spell-checkers** The database for a spell-checker might be a list of words, one per line, the minimal structure required by the \texttt{buftable} class. Some classes introduce new terminology rather than to define a new data structure. This example introduces a lookup operation that can fail, for use in tests. In addition, since many spell-checking systems allow user definable dictionaries in addition to their central database, \texttt{spellChecker} objects may chain together for the purpose of looking up words.

\begin{verbatim}
class spellChecker : buftable(parent)
  method spell(s)
    return \text{\texttt{\textbackslash text}[s]} | (\texttt{\textbackslash parent}).spell(s)
  end
end
\end{verbatim}

**Dictionaries** Dictionaries are slightly more involved. Each entry might consist of a part of speech, an etymology, and an arbitrary string of uninterpreted text comprising a definition for that entry, separated by semicolons. Since each such entry is itself a structure, a sensible decomposition of the dictionary structure consists of two classes: one that manages the table and external file I/O, and one that handles the manipulation of dictionary entries, including their decoding and encoding as strings.

\begin{verbatim}
class dictionaryentry(word, partofspeech, etymology, definition)
  # decode a dictionary entry into its components
  method decode(s)
    s ? {
      word := tab(upto(';'))
      move(1)
      partofspeech := tab(upto(';'))
      move(1)
      etymology := tab(upto(';'))
      move(1)
      definition := tab(0)
    }
\end{verbatim}
11.1. INHERITANCE

```unicon
method encode() # encode a dictionary entry into a string
    return word || ";" || partofspeech || ";" || etymology || ";" || definition
end

initially
    if /partofspeech then {
        # constructor was called with a single string argument
        decode(word)
    }
end

class dictionary : buftable()
method read()
    self.buffer.read()
    tmp := table()
    every line := !text do
        line ? { tmp[tab(many(&letters))] := dictionaryentry(line) | fail }
    text := tmp
end

method write()
    f := open(filename, "w") | fail
    every write(f, (!text).encode)
    close(f)
end
```

Thesauri Although an oversimplification, one might conceive of a thesaurus as a list of entries, each of which consists of a comma-separated list of synonyms followed by a comma-separated list of antonyms, with a semicolon separating the two lists. Since the code for such a structure is nearly identical to that given for dictionaries above, we omit it here (you might start by generalizing class `dictionaryentry` to handle arbitrary strings organized as fields separated by semicolons).

A (toy) Dictionary Program

The above examples were intended to illustrate a pedagogical point about how inheritance facilitates specialization of existing code. Although they are legal Unicon code, they were always toy examples that illustrate ideas, not an extract from a real software application. Consult the Unicon translator implementation, or the Unicon GUI classes, for many real-world examples of inheritance.

Having said that, here is a complete program that makes use of some of the preceding class examples. The program, called deen (from Deutsch-English) reads in a dictionary in
plain text format, originally obtained from http://ftp.tu-chemnitz.de/pub/Local/urz/ding/de-en/ and writes out English entries for one or more German language words given on the command-line. German was chosen fairly arbitrarily here, but the code does depend on the file format of the dictionary file, de-en.txt.

# deen.icn - give English equivalents of German words
#

procedure main(av)
  if av[1]="-all" then all := pop(av)
  dd := DeenDictionary()
  every s := !av do
    if lu := dd.lookup(s) then {
      if \all then
        every write(s, ": ", dd.lookup(s).definition)
      else write(s, ": ", lu.definition)
    }
    else write(s, " is not in the dictionary.")
end

class buffer(filename, text)
  # read a buffer in from a file
  # todo: decompress if .gz extension
  method read()
    if match("http://", filename) then mode := "m" else mode := "r"
    f := open(filename, mode) | stop("can't open ", image(filename))
    if filename[-3:0] == ".gz" then {
      if mode=="m" then { # download_to_local_file
        if not (f2 := open("de-en.txt.gz","w")) then
          stop("can't write")
        while s := reads(f, 1000000) do writes(f2, s)
        close(f)
      }
      system("gunzip de-en.txt.gz")
      f := open("de-en.txt") | stop("can't read de-en.txt")
    }
    writes("Opened ",image(f),".\nReading")
    text := [ ]
    every put(text, f) do if *text%1000=0 then writes("")
close(f)
write("\ndone. Read ", *text, " lines")
return
end
method erase()
#... ?
end
#... additional buffer operations
initially
if \filename then read()
end
class buftable : buffer()
method read()
  self.buffer.read()
  tmp := table()
  every line := !text do {
    line ? {
      word := tab(many(&letters)) | stop("failed on ", image(line))
      tmp[word] := line
    }
  }
  text := tmp
  return
end
method lookup(s)
  suspend ! \ (text[s])
end
class dictionaryentry(word, part, etymology, definition)
  # decode a dictionary entry into its components
  # assumed format is word;pos;eym;def
method decode(s)
  s ? {
    word := tab(find(";"))
    move(1)
    part := tab(find(";"))
    move(1)
    etymology := tab(find(";"))
    move(1)
    definition := tab(0)
  }
end
method encode() # encode a dictionary entry into a string
    return word || ";" || part || ";" || etymology || ";" || definition
end

initially
    if /part then # constructor was called with a single string argument
        decode(word)
    end

class dictionary : buftable()
    method read()
        self.buffer.read()
        tmp := table()
        every line := !text do
            line ? {
                word := tab(many(&letters)) | stop("failed on ", image(line))
                tmp[word] := dictionaryentry(line) | fail
            }
        text := tmp
    end
    method Write()
        f := open(filename, "w") | fail
        every write(f, (!text).encode)
        close(f)
    end
end

class DeenEntry : dictionaryentry(gender)
    initially(de, en)

    de ? {
        if word := trim(tab(find("\"\"\"\"\")\"),0) then {
            ="\"\"
            gender := tab(find("\"\"\"\"\")\"")
        }
        else { # here is one without gender info
            word := trim(tab(find("\"\")\"),0,0)
            gender := "?"
        }
    }
    definition := en
end

# # Return a list of dictionary entries for a given line of text
# procedure get_entries(s)
subentries := []
s ? {

deutsch := tab(find("::")) | stop("no :: in ", image(s))
="::"; tab(many(' '))
english := tab(0)

deutsch ? {
    while *(deutschwort := tab(find("|") | 0))>0 do {
        deutschwort := trim(deutschwort,0)
    ="|"
        tab(many(' '))
        englishword := trim(english[1:find("|",english)|0],0)
        english ?:= {
            tab(many(' \t'))
            =englishword
            tab(many(' \t'))
            ="|
            tab(many(' \t'))
            tab(0)
        }
    if i := find(";", deutschwort) then {
        deutschwort ? {
            while *(dword := tab(find(";") | 0))>0 do {
                =","
                tab(many(' '))
                if gronk:=englishword[1:upto(';', englishword)|0] then {
                    if *gronk>0 then {
                        eword := gronk
                    }
                }
                put(subentries, DeenEntry(dword, eword))
                englishword ?:= { =eword; =","; tab(many(' ')); tab(0)}
            }
        }
        else {
            put(subentries, DeenEntry(deutschwort, englishword))
        }
    }
    }
else {
    put(subentries, DeenEntry(deutschwort, englishword))
}
}
return subentries
198  

**CHAPTER 11. INHERITANCE AND ASSOCIATIONS**

```unicon
end

class DeenDictionary : dictionary()
  method read()
    self.buffer.read()
    tmp := table()
    every line := !text do
      line ? {
        if ="#" | line=="" then next
        if not (L := get_entries(line)) then
          stop("get_entries failing on ", image(line))
        every x := !L do {
          if not member(tmp, x.word) then
            tmp[x.word] := [x]
          else put(tmp[x.word], x)
        }
      }
    text := tmp
  end
initially
  if stat("de-en.txt") then filename := "de-en.txt"
  else if stat("de-en.txt.gz") then filename := "de-en.txt.gz"
  else filename := DEEN
  self.read()
end
```

**Superclass cycles and type equivalence**

In many situations, there are several ways to represent the same abstract type. Two-dimensional points might be represented by Cartesian coordinates \(x\) and \(y\), or equivalently by radial coordinates expressed as a distance \(d\) and angle \(r\) given in radians. If one implements classes corresponding to these types there is no reason one of them should be considered a subclass of the other; they are interchangeable and equivalent.

In Unicon, expressing this equivalence is simple and direct. In defining classes **Cartesian** and **Radial** we may declare them to be superclasses of each other:

```unicon
class Cartesian : Radial (x, y)
  # code that manipulates objects using Cartesian coordinates
end

class Radial : Cartesian (d, r)
  # code that manipulates objects using radial coordinates
end
```
11.2 Associations

An association denotes a relationship between classes that occurs between specific instances of the classes at runtime. Just as objects are instances of classes, associations have instances called links. Like objects, links have a lifetime, from the instant at which the relationship is established to the time at which the relationship is dissolved. Besides serving to connect two objects, links may have additional state information or behavior; in the most general case links can be considered to be objects themselves: special objects whose primary role is to interconnect other objects.

11.3 Aggregation

Inheritance may be the most famous kind of relationship between classes, but it is not the most indispensable. Many languages that provide objects do not even bother with inheritance. The composition of assembly objects from their component parts, on the other hand, is a truly ubiquitous and essential idea called aggregation. The class dictionary defined in the previous section is a good example of an aggregate object; its component parts are dictionaryentry objects.

Aggregation is depicted in class diagrams by a line between classes with a diamond marking the aggregate, or assembly, class. Figure 11-3 shows an aggregate found in the domain of sports, where a team is comprised of a group of players.
Unlike inheritance, aggregation describes a relationship between instances at run-time. Different aggregate instances are assembled from different component part instances. While a given player might play for different teams over a period of time, at any given instant a player is normally part of at most one team.

### 11.4 User-defined associations

All the other relationships between classes in UML are left to the application designer to specify as custom associations. User-defined associations are depicted by lines, annotated with the association name next to the line, midway between the two classes. Figure 11-4 shows a silly user-defined association that describes a family relationship called Marriage. For a diagram containing such an association to be well defined, the semantics of such a relationship must be specified in external documentation that describes Marriage for the purposes of the application at hand.

![Figure 11-4: A User-Defined Association](image)

### Multiplicities, roles, and qualifiers

Whether they are aggregations or user-defined application domain relationships, associations are not completely specified until additional details are determined during program design, such as how many instances of each type of object may participate in a given relationship. These additional details appear in canonical positions relative to the line that depicts the association within a class diagram.

A *multiplicity* is a number or range that indicates how many instances of a class are involved in the links for an association. In the absence of multiplicity information an association is interpreted as involving just one instance. It normally appears just below the association line next to the class to which it applies. Figure 11-5 shows a BasketballTeam that is an aggregate with a multiplicity of five Players.

![Multiplicities, roles, and qualifiers](image)
A multiplicity range is expressed as a pair of numbers separated by two periods, as in 1..3. The value * may be used by itself to indicate that a link may associate any number (zero or more) of objects. The * value may also be used in a range expression to indicate no upper bound is present, as in the range 2..*.

A role is a name used to distinguish participants and responsibilities within an association. Roles are drawn just above or below the association line, adjacent to the class to which they refer. They are especially useful if a given association may link multiple instances of the same class in asymmetric relationships. Figure 11-6 shows a better model of the classes and roles involved in a Marriage association depicted earlier in Figure 11-4.

A qualifier is a key value used to distinguish instances in a link, in lieu of a large multiplicity that would otherwise be inefficient. For example, a directory may contain many files, but each one may be directly accessed by name. A qualifier is drawn as a rectangular association end with the qualifier key given inside the rectangle. Figure 11-7 shows a reinterpretation of the basketball aggregation in which the players on the team are distinguished using a qualifier key named position.

Implementing associations

Unlike inheritance, which is implemented by the language and resolved at compile time, associations involve dynamic relationships established at runtime, and are implemented by the programmer...or are they? In the most general case, an association may be implemented by writing a class whose instances are links between the related classes’ objects. In the
narrowest special case, an association can be implemented by adding an attribute in one class to contain a reference to an object in an associated class. Much of the value introduced by multiplicity and qualifier information is to narrow the association semantics down to what is readily implementable using the built-in structure types instead of writing classes for them. If an association can be implemented using a list or table instead of defining a new class, the resulting code will be smaller and faster.

In all cases, associations will introduce additional fields into the classes being associated. The following code example implements the Marriage association from Figure 11-6. For illustration purposes it is modeled as a one-one relationship at any given point in time. Notice the intertwining methods in the two classes that establish a bi-directional relationship. Error checking is left as an exercise for the reader.

```plaintext
class Man(wife)
    method Marry(w)
        wife := w
        if not (self === w.husband) then w.Marry(self)
    end
end

class Woman(husband)
    method Marry(m)
        husband := m
        if not (self === m.wife) then m.Marry(self)
    end
end
```

As a general rule, an association that has a qualifier is implemented with a table. The following example corresponds to the basketball team diagram in Figure 11-7. The players attribute might be a list or set in class Team, but the qualifier allows class BasketballTeam to override this and implement players using a table. Such a refinement can be awkward in a statically typed object-oriented language. Depending on whether its player parameter is supplied or is null, method Player() serves to either lookup a player, given a position, or to insert a player into the association. In either case the player at the designated position is returned.

```plaintext
class BasketballTeam : Team ()
    method Player(position, player)
        players[position] := \player
        return players[position]
    end
    initially
        players := table()
end
```
Associations with multiplicity might be implemented using sets or lists, with lists being favored when multiplicity is bounded to some range, or when there is a natural ordering among the instances. The following version of the BasketballTeam class uses a list of five elements to implement its aggregation, which occupies less space than either a set or the table in the last example.

```unicon
class BasketballTeam : Team (players)
    method Player(player)
        if player === !players then fail # already on the team
        if /!players := player then return # added at null slot
        ?players := player # kick someone else off team to add
    end

initially
    players := list(5)
end
```

Defining a new class to implement an association handles rare cases such as many-many relationships and associations that have their own state or behavior. Other examples of associations and their implementation are given in Part 3 of this book.

### 11.5 Summary

The relationships between classes are essential aspects of the application domain that are modeled in object-oriented programs. You can think of them as the "glue" that connects ideas in a piece of software. Class diagrams allow many details of such relationships to be specified graphically during design. Unicon’s structure types allow most associations to map very naturally onto code. In order to understand the subtleties of how these features are implemented, you may wish to study the output of `unicon -E` for various examples; the `-E` option writes out the Icon translation of the object-oriented code.
Chapter 12

Writing Large Programs

This chapter describes language features, techniques, and software tools that play a sup-
porting role in developing large programs and libraries of reusable code. You can write large
programs or libraries without these tools; the tools just make certain tasks easier. These
facilities are not specific to object-oriented programs. However, they serve the same target
audience since one of the primary benefits of object-orientation is to reduce the difficulty
of developing large programs.

In the case of Unicon, "large" might mean any program so complex that it is not self-
explanatory without any special effort. This includes all programs over a few thousand lines
in size, as well as shorter programs with complex application domains, and those written
by multiple authors or maintained by persons other than the original author.

Writing and maintaining large programs poses additional challenges not encountered
when writing small programs. The need for design and documentation is greater, and the
challenge of maintaining the correspondence between design documents and code is more
difficult. Design patterns can help you with the design process, since they introduce easily
recognizable idioms within a design. The more familiar the design is, the less cognitive load
is imposed by the task of understanding it.

This chapter shows you how to:

• Understand the difference between abstract and concrete classes
• Use design patterns to simplify and improve software designs
• Organize programs and libraries into packages
• Generate HTML indices and reference documentation for your code

12.1 Abstract Classes

In programming languages, it turns out there are at least two very different kinds of things
referred to by the word "class". Most classes denote a data type, of which there are one or
more instances. The word class is also used to denote a general category of objects, of which there are no actual instances. Such classes are called abstract classes. Abstract classes are used by defining subclasses which do have instances. In a small program you might not run into a need for abstract classes, but if you write a large object-oriented program, or your small program uses someone else’s large object-oriented class library, you are likely to need to understand abstract classes.

The whole idea of class and inheritance originated by analogy from biology, but if you think about it, biology uses a host of different terms to denote categories (phylum, family, genus, etc.), that are distinct from the term that denotes a class of instances (species). There are plenty of instances of species Homo sapiens, but there are no instances of a genus Homo, there are only instances of Homo’s subclasses. Similarly, if you are designing software to model the behavior of cars and trucks, you may identify a lot of shared behavior and place code for it in a superclass Vehicle, but there are no instances of Vehicle, only its subclasses.

In a larger inheritance graph (or tree) most classes may well be abstract. It may be those classes (primarily leaves) of which there are actual instances that are special and deserve a different term besides "class". We could easily think of all classes as abstract by default, and refer to classes that have instances as "concrete classes" or "instantiation classes". The analogy to biology would be better served by such terminology. But for better or for worse, most software engineers will have to live with "abstract class" and "class" to denote general categories and instantiable categories, respectively.

Numerous larger and more concrete examples of abstract classes appear in the Unicon GUI class library, described in Chapter 17. Some classes, such as Button, denote general categories of widgets that have code and behavior in common (such as the fact that you can click them). There are no instances of Button, there are only subclasses such as TextButton, IconButton, and CheckBox, that may be instantiated. In larger applications, the code sharing (or duplication avoidance) of abstract classes may be compelling, but like all uses of inheritance they do require you to read and understand multiple bodies of code (the class and all its superclasses) in order to use the class effectively.

12.2 Design Patterns

Class and function libraries provide good mechanisms for code reuse, and inheritance helps with code reuse in some situations. But in large programs it is desirable to reuse not just code, but also successful designs that capture the relationships between classes. Such design reuse is obtained by identifying design patterns with known successful uses in a broad range of application domains. For example, the practice of using pipes to compose complex filters from more primitive operations has been successfully used in compilers, operating system shells, image processing, and many other application areas. Every programmer should be familiar with this pattern.
The field of software design patterns still quite young. Practitioners are producing simple catalogs of patterns, such as the book *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. When the field is more mature it will include syntactic and/or semantic rules for how patterns are combined to form higher-order patterns, as is the case for building architecture. This section presents a few classic patterns and their implementation in Unicon.

At best, this discussion of patterns may whet your appetite to go read more about the subject. In addition to their design reuse value, patterns also provide software developers with a common vocabulary for discussing recurring design concepts. The judicious use of one or more abstract classes seems to be a recurring theme throughout most of the design patterns identified by Gamma et al.

**Singleton**

Perhaps the simplest design pattern is the singleton, describing a class of which exactly one instance is required. Singletons are interesting because they are related to *packages*, a language feature described later in this chapter. A package is a mechanism for segregating a group of global objects so that their names do not conflict with the rest of the program. This segregation is similar to that provided by object encapsulation; a package is similar to a class with only one instance.

Consider as an example a global table that holds all the records about different employees at a small company. There are many instances of class Employee, but only one instance of class EmployeeTable. What is a good name for this instance of EmployeeTable, and how can you prevent a second or subsequent instance from being created? The purpose of the singleton pattern is to answer these questions.

In Unicon, one interesting implementation of a singleton is to replace the constructor procedure (a global variable) by the instance. Assigning an object instance to the variable that used to hold the constructor procedure allows you to refer to the instance by the name of the singleton class. It also renders the constructor procedure inaccessible from that point on in the program’s execution, ensuring only one instance will be created.

```unicon
class EmployeeTable(...)
  initially
    EmployeeTable := self
  end
```

There are undoubtedly other ways to implement singleton classes.

**Proxy**

A proxy is a "stand-in" for an object. The proxy keeps a reference to the object it is replacing, and implements the same interface as that object by calling the object’s version
of the corresponding method each time one of its methods is invoked. Figure 12-1 shows a proxy serving a client object in lieu of the real object.

![Figure 12-1: The Proxy Pattern](image)

Proxies are used when the original object cannot or should not be invoked directly. If the object is on a remote machine, the proxy can take care of network communication and hide the location of the object from the clients. As another example, the original object may be instantiated lazily - it may be a large object that is not loaded into memory unless one of its operations is invoked.

Similar to both of the preceding examples, mobile objects might be implemented using proxies. If several machines are running a computation jointly and communicating, some gigantic object might be kept on one machine at a time. In applications with strong locality of reference, whenever a machine needs to do a call on the gigantic object it might do hundreds of calls on that object. In that case the object should move from wherever it is, to the machine where it is needed. The rest of the program does not need to be aware of whether the object is local, remote, or mobile; it just interacts with the proxy instance.

```plaintext
class gigantic(x1,x2,...,x1000)
    method invoke()
       ...
    end
initially
    # Gigantic object's state is loaded from network
end
class proxy(g)
    method invoke()
       /g := gigantic()
       return g.invoke()
```
Chain of responsibility

This pattern is similar to a proxy, in that an object is delegating one or more of its methods to a second object. It is not presented in detail, but its similarity to proxies is mentioned because many design patterns in the Gamma book seem incredibly similar to each other; reading the book is like having déjà vu all over again. Perhaps there ought to be some kind of orthogonality law when it comes to patterns.

The difference between a chain of responsibility and a proxy is that the proxy forwards all method invocations to the "real" object, while in a chain of responsibility, the object may handle some methods locally, and only delegate certain methods to the next object in the chain. Also, proxies are normally thought of as a single level of indirection, while the chain of responsibility typically involves multiple linked objects that jointly provide a set of methods. The following example illustrates a chain of responsibility between a data structure object (a cache) and an Image class that knows how to perform a computationally intensive resolution enhancement algorithm.

class Image(...)
    method enhance_resolution(area)
        # enormous computation...
    end

initially
    # ... lots of computation to initialize lots of fields
end

class hirez_cache(s, t)
    method enhance_resolution(area)
        if member(t,area) then { # proxy handles
            return t[area]
        }
        # else create the gigantic instance
        /im := image()
        return t[area] := im.enhance_resolution(area)
    end

initially
    t := table()
    # Insert some known values for otherwise enormous computation.
The instance of class Image is not created until one is needed, and image’s method 
`enhance_resolution()` is not invoked for previously discovered results. Of course, 
`enhance_resolution()` must be a pure mathematical function that does not have any side 
effects for this caching of results to be valid.

**Visitor**

The visitor pattern is a classic exercise in generic algorithms. It is fairly common to have a 
structure to traverse, and an operation to be performed on each element of the structure. 
Writing the code for such a traversal is the subject of many data structure texts. In fact, 
if you have one operation that involves traversing a structure, there is a good chance that 
you have (or will someday need) more than one operation to perform for which the same 
traversal is used. Figure 12-2 illustrates the visitor pattern.

![Visitor Pattern Diagram](image)

**Figure 12-2: The Visitor Pattern**

The visitor pattern says you can separate out the traversal algorithm from the operation 
performed at each element of the structure, and reuse the traversal on other operations. The 
following code illustrates this separation of traversal (implemented by method `Accept()`) 
from visitation (implemented by methods `DoLeaf()` and `DoInternalNode()` in the visitor). 
Where there is one kind of visitor there may be many, and in that case, class Visitor 
may be an abstract class, instantiated by many concrete Visitor subclasses that have the 
same method names but do not share code. Note also that this code example allows for 
heterogeneous structures: the Visitor just defines a "Do..." method for each type of node in 
the structure.

```plaintext
class Visitor()
```
method DoLeaf(theLeaf)
    # ... visit/use theLeaf.datum
end
method DoInternalNode(theNode)
    # ... visit/use theNode.datum
end

class Leaf(datum)
    method Accept(v)
        v.DoLeaf(self)
    end
end
class InternalNode : Leaf(children)
    method Accept(v)
        every (!children).Accept(v)
        v.DoInternalNode(self)
    end
end

Executing a traversal from a root object looks like root.Accept(myvisitor) where myvisitor is an instance of some Visitor class. The point of the Visitor pattern is that you can define different Visitor classes. For example, here are Visitors to print a tree, and to calculate and store the heights of all nodes in the tree:

class Printer()
    method DoLeaf(theLeaf)
        writes(theLeaf.datum, " ")
    end
    method DoInternalNode(theNode)
        writes(theNode.datum, " ")
    end
end
class Heights()
    method DoLeaf(theLeaf)
        theLeaf.datum := 1
    end
    method DoInternalNode(theNode)
        theNode.datum := 0
        every theNode.datum <=: (!children).datum
        theNode.datum +=: 1
    end
end
12.3 Packages

In large programs, the global name space becomes crowded. You can create a disaster if one of your undeclared local variables uses the same name as a built-in function, but at least you can memorize the names of all the built-in functions and avoid them. Memorization is no longer an option after you add in hundreds of global names from unfamiliar code libraries. You may accidentally overwrite some other programmer’s global variable, without any clue that it happened.

Packages allow you to partition and protect the global name space. A package is similar to a “singleton” class with only one instance. Every global declaration (variables, procedures, records, and classes) is "invisible" outside the package, unless imported explicitly.

The package declaration

A package declaration specifies that all global symbols within a source file belongs to a package. The package declaration looks similar to the link declaration. You provide the package name, an identifier, or a string filename:

    package foo

or

    package "/usr/local/lib/icon/foo"

There can be only one package declaration in a source file. It need not be at the beginning of the source file, but this is conventional. Within a package, global names defined inside the package are referenced normally. Global names outside the package are not visible by default. Here is an example source file that declares some globals and adds them to a package.

```plaintext
# pack1.icn
package first
procedure my_proc()
    write("In my_proc")
end
class SomeClass()
    method f()
        write("In SomeClass.f")
    end
end
```
When this code is compiled, the information that package first contains the symbols my_proc and SomeClass is recorded into a database and that using package first implies linking in pack1.u along with any other files that are part of package first. In order to prevent name conflicts the compiler also applies a name mangling process to the global symbols, described below.

The import declaration

To access symbols within another package, use the import declaration, which has the following syntax:

```plaintext
import foo
```

This causes the compiler to look up the package in its database and identify its symbols. Import declarations use the IPATH environment variable in the same way as do link declarations. In particular, an import declaration is a link declaration, augmented with scope information about the names defined in the package.

Explicit package references

Sometimes, two imported packages may define the same symbol, or an imported symbol conflicts with a global declaration in one of your files. To resolve these problems, you can explicitly specify the package to use for particular symbol references. For example, if packages first and second both define a procedure named write, then

```plaintext
import first, second
procedure main()
    first::write()  # calls write() in package first
    second::write() # calls write() in package second
    ::write()      # calls the global write()
end
```

The use of the :: operator on its own is a useful way to refer to a global procedure from within a class that has a method of the same name, as in

```plaintext
class Abc(x)
    method write()
        ::write("Abc x=", x)
    end
end
```

In this example, omitting the :: would cause the write() method to call itself until the program runs out of memory and produces a runtime error.
Name conflicts and name mangling

The purpose of packages is to reduce name conflicts, especially accidental ones. You will get a link error if you declare the same name twice in the same package. You will get a compile error if you try to import a package that contains a variable that is already declared. In Unicon, unlike Arizona Icon, you will also get a warning message if you declare a global variable of the same name as a built-in function, or assign a new value to such a name. Often this is done on purpose, and it shows off the flexibility of the language. But other times when it happens by accident, it is a disaster. Such warnings can be turned off with the -n option to the unicon compiler.

Under the hood, packages are implemented by simple name mangling that prefixes the package name and a pair of underscores onto the front of the declared name. You can easily defeat the package mechanism if you try, but the reason to mention the name mangling is so you can avoid variable names that look like names from other packages.

A similar name mangling constraint applies to classes. Also, the compiler reserves field names __s and __m for internal use; they are not legal class field names. Identifiers consisting of __n, where n is an integer are reserved for Unicon temporary variable names. Finally, for each class foo declared in the user’s code, the names foo, foo__state, foo__methods, and foo__oprec are reserved, as are the names foo_bar corresponding to each method bar in class foo.

Compilation order and the unidep tool

When possible, you should compile all files in a package before you import that package. Even if you do, if multiple source files belong to the same package, the order in which they are compiled is significant. Consider the following code in three source files:

```plaintext
# order1.icn
package demo
procedure first()
    write("first")
end

# order2.icn
package demo
procedure second()
    write("second")
    first()
end

# order3.icn
import demo
procedure main()
```
second()
end

Files order1.icn and order2.icn belong to a package demo, which is used by order3.icn. You can rightly guess that order3.icn should be compiled after order1.icn and order2.icn, but does it matter which of them is compiled first? If order2.icn is compiled first, Unicon’s database does not know symbol first is part of the package, and does not mangle the name; if you compile these files out of order you will get a runtime error.

The brute force solutions you have available to you are: to always place all of a package in the same source file, or to compile the files twice. Neither of these options is especially appealing. The symbol references in each package’s files form a graph of dependencies on the other files in the same package. As long as this graph is acyclic, a correct order can be calculated. Unidep is a program that automates this task and generates a makefile specifying the dependencies in build rules. For example, given the program above, and the following makefile:

```plaintext
order: order1.u order2.u order3.u
    unicon -o order order1.u order2.u order3.u
%.u: %.icn
    unicon -c $*
```

Running the command “unidep order1.icn order2.icn order3.icn” will append the required additional dependencies. In this case these are:

```plaintext
order1.u: order1.icn
order2.u: order2.icn order1.u
order3.u: order3.icn order2.u
```

With these dependencies added, the makefile will compile the files in the correct order. You will want to add a rule to invoke Unidep from the makefile, and rerun it when your program changes significantly.

12.4 HTML documentation

Iplweb is an Icon documentation generator, inspired loosely by Java’s JavaDoc program, and based on an HTML-generating program called iplref, by Justin Kolb. Iplweb depends on your program being in “IPL normal form”, which is to say that comments in your source files should be in the format used in the Icon Program Library. From these comments and the signatures of procedures, methods, records, and classes, Iplweb generates reference documentation in HTML format.
This approach produces reference documentation automatically, without altering the original source files. Run Iplweb early, and run it often. It is common for reference documentation to diverge over time from the source code without such a tool. It is especially suitable for documenting the interfaces of procedure and class libraries. What it doesn’t help with is the documentation of how something is implemented. It is designed primarily for the users, and not the maintainers, of library code.

12.5 Summary

Writing and maintaining large programs poses additional challenges not encountered when writing small programs. The need for design and documentation is greater, and the challenge of maintaining the correspondence between design documents and code is more difficult. Design patterns can help you with the design process, since they introduce easily recognizable idioms or sentences within a design. The more familiar the design is, the less cognitive load is imposed by the task of understanding it.

Packages have little or nothing to do with design patterns, but they are just as valuable in reducing the cognitive load required to work with a large program. Packages are not just a source code construct. They actually do play a prominent role in software design notations such as UML. From a programmer’s point of view, packages protect a set of names so that their associated code is more reusable, without fear of conflicts from other reusable code libraries or the application code itself.
Chapter 13

Use Cases and Supplemental UML Diagrams

When starting a new software project, it is tempting to begin coding immediately. An advocate of stepwise refinement starts with the procedure `main()` that every program has, and grows the program gradually by elaboration from that point. For complex systems, a software designer should do more planning than this. Chapters 9 and 10 covered the basics of class diagramming, an activity that allows you to plan out your data structures and their interrelationships.

The hard part about class diagramming is figuring out what information will need to be stored in attributes, and what object behavior will need to be implemented by methods. For many large projects there are basic questions about what the program is supposed to do that must be answered before these details about the application’s classes can be determined. In addition, class diagrams depict static information but model nothing about the system that involves changes over time.

This chapter discusses some UML diagramming techniques that are useful before you start coding. They can help you figure out the details that belong in your class diagrams, by modeling dynamic aspects of your application’s behavior. When you are finished with this chapter you will know how to:

- Draw use case diagrams that show the relationships between different kinds of users and the tasks for which they will use the software.
- Describe the details of use cases that define an application’s tasks.
- Draw statechart diagrams that depict an object’s behavior as states and transitions between states that model the dynamic aspects of the application.
- Specify conditions and activities that occur when an event causes an object to change its state.
• Draw collaboration diagrams that illustrate dynamic interactions between groups of objects.

13.1 Use Cases

A use case is an individual task. It defines a unit of functionality that the software enables one or more users to carry out. Sometimes it is a challenge to figure out what makes a reasonable “unit of functionality” in an application where long sequences of complex tasks are performed. Should the use cases correspond to small units such as individual user actions such as mouse clicks, or longer jobs such as updating a spreadsheet? One way to identify the appropriate units of functionality is to ask, for any given user action, whether it completes a change to the state of the application data. If the user would likely want to be able to save their work afterwards, the task is large enough to constitute a use case.

A diagram showing all the use cases helps early on in development to identify the overall scope and functionality of the software system as seen from the outside. The components of a use case diagram are depicted in Figure 13-1.

![Use Case Diagram Components](image)

Figure 13-1: The main components of use case diagrams

The use cases themselves are shown as ovals. The name of the use case is inside the oval. The use cases have an accompanying description; an example description is given in
the next section. A use case is not represented in software by a class, but rather in the logic of the program’s control flow. A use case relates several otherwise unassociated objects for a limited time to accomplish a particular task.

The term *actor* denotes both human users and external hardware or software systems that interact with the software system under design. Actors are shown in use case diagrams as stick figures. Each stick figure in the diagram represents a different kind of actor that interacts with the system during one or more use cases. The name of the role is written under the stick figure. An actor is really just a special kind of class that represents an external, asynchronous entity.

The *associations* between use cases and the actors that perform those tasks are drawn as plain lines. A use case may be performed by one or several actors. Use case associations identify the actors that participate in each use case. They are only slightly related to the associations between classes found in class diagrams.

*Dependencies* and *elaborations* between use cases are drawn as lines with arrows, annotated with a label between « and ». Some use cases use other use cases as part of a more complex task. Other use cases are defined as extensions of another use case.

**Use case diagrams**

A use case diagram consists of a set of use case ovals, bordered by a rectangle that signifies the extent of the software system. Actors are drawn outside the rectangle, with connecting lines to those use cases in which they participate. When some actors are non-human external systems, by convention the human actors are depicted on the left, and the non-humans go on the right.

An example use case diagram is shown in Figure 13-2, which depicts a recruiting management system. The manager hiring a new employee may interact with the company’s legal department to produce an acceptable position advertisement. Many applicants might apply for a given position. The manager evaluates applications, possibly interviewing several candidates. When a candidate is selected, the manager interacts with the legal department to make a job offer.
Use case descriptions

The details of each use case are specified in a related *use case description*. This description may include prose text, such as the following description of the “Make Offer” use case:

**Make Offer** is started by the manager when an applicant has been selected from among the candidates for a position. The manager obtains approval from the legal department, commits necessary budget resources, and generates an offer letter with details on salary, benefits, and the time frame in which a decision is required.

The use case description may be organized into fields, or more detailed than this. For example, one field might consist of the most common sequence of events, emphasized by an explicit enumeration. The common variations on the primary event sequence are also of value. A more organized description of the Make Offer use case might be

**Make Offer Initiated**: by manager, after candidate for a position has been selected.

**Terminates**: when the candidate receives the offer in writing.

Sequence:
1. Manager obtains approval from legal department.
2. Manager commits resources from budget
3. Manager telephones candidate with offer
4. Manager generates offer letter
5. Offer letter is express mailed to candidate.

Alternatives:
- In step 2, Manager may request extra non-budgeted resources.
- In step 3, Manager may fax or e-mail offer in lieu of telephone.
13.2 Statechart Diagrams

Statecharts are diagrams that depict finite state machines. A finite state machine is a set of states, drawn as circles or ovals, plus a set of transitions, drawn as lines that connect states. Statecharts generally have an initial state, which may be specially designated by a small, solid circle, and one or more final states, which are marked by double rings.

In object modeling, states represent the values of one or more attributes within an object. Transitions define the circumstances or events that cause one state to change to another. Statecharts are a tool for describing allowable sequences of user interactions more precisely than is captured by use cases. Discovering the events that cause transitions between states, as well as the conditions and actions associated with them, helps the software designer to define the required set of operations for classes.

Figure 13-3 shows an example statechart diagram for a real estate application. A house enters the FORSALE state when a listing agreement is signed. The house could leave the FORSALE state with a successful offer at the listed price (entering a REVIEW period) or by utter failure (if the listing agreement expires), but the most common occurrence is for a buyer to make an offer that is less than the asking price. In that case, a NEGOTIATION state is entered, which may iterate indefinitely, terminating when either the buyer or seller agrees to the other party’s offer or walks away from the discussion. When an offer is accepted, a PENDING period is entered in which financing is arranged and inspections and walkthroughs are performed; this period is terminated when escrow is closed, title is transferred, and the house is SOLD.

Figure 13-3: A Statechart Diagram

Since a state represents the values of one or more attributes within an object, a transition coincides with assignments that alter those attributes’ values. The purpose of the diagram is to identify when and why those values change, in terms of the application domain.
Events and conditions

Most transitions in a statechart are triggered by an event. In Figure 13-3 the events were things like “offer” and “closing”. Typically, an event describes an asynchronous communication received from another object. An event is instantaneous, while a state corresponds to some possibly lengthy interval of time until an object transitions into some other state. From the point of view of the object being modeled in the statechart, the event is an interrupt that affects the object’s behavior. Such an event would normally be implemented by defining a method for the object with a name derived from the event name.

It is common during modeling to have a transition that can only occur if a Boolean condition is satisfied. In Figure 13-3, the event offer was used for two transitions out of the same state, with different conditions (amount $\geq$ list price versus amount $< list price) to determine which transition would be taken. In statechart diagrams, conditions are given after the event name, in square brackets, as in [amt < list].

For a condition on a transition, it might make sense for that transition to require no trigger event at all. The transition would occur immediately if the condition were ever satisfied. Such a constraint-based transition would potentially introduce condition tests at every point in the object’s code where the condition could become true, such as after each assignment to a variable referenced in the condition. This may work in special cases, but poses efficiency problems in general. Transitions without trigger events make sense in one other situation. If a state exits when a particular computation completes, you can use a triggerless transition to the new state that the object will be in when it is finished with the job it is performing in the current state.

Actions and activities

Events are not the only class methods that are commonly introduced in statecharts. In addition to a condition, each event can have an associated action. An action is a method that is called when the event occurs. Since events are instantaneous, action methods should be of bounded duration. Similarly, states can have a whole regalia of related methods called activities. There are activities that are called when a state is entered or exited, respectively. The most common type of activity is a method that executes continuously as long as the object is in that state. If more than one such activity is present, the object has internal concurrency within that particular state.

In statechart diagrams, actions are indicated by appending a slash (/) and an action after the event name and any condition. Activities are listed within the state oval. If a keyword and a slash prefix the activity, special semantics are indicated. For example, the do keyword indicates repeated activity. In Figure 13-3, the activity do / show() says that the house will be shown repeatedly while it is in the FORSALE state. The activity entry / open_escrow() indicates that the method open_escrow() is called on entry to the PENDING state, after which inspections() and arrange_financing() activities are performed.
13.3 Collaboration Diagrams

Statecharts normally model the state of one object. They show how the object reacts to events that come from the other objects in the system, but do not depict where those events came from. In a complex system, it is useful to understand the interactions among many objects. An event that changes one object’s state may trigger events in many other objects, or a group of objects may trigger events in one another in a cyclic fashion.

Collaboration diagrams show such interactions between objects. They are drawn similarly to class diagrams. A group of rectangles are drawn to represent instances of classes, and lines depict the relationships between those classes. But while a class diagram emphasizes the static structures, representing details such as class attributes, and association multiplicity, a collaboration diagram depicts a specific sequence of messages sent from object to object during the completion of some task. The messages are annotated alongside the links between objects to indicate sender and recipient, and numbered to show both the sequence and the tree structure of the nested messages. In the general case more than one message number can be annotated for a given link, since multiple messages may be transmitted between the same objects in the course of completing the use case.

Figure 13-4 shows an example collaboration diagram. This particular collaboration illustrates the input processing of a user event in a game application in which pieces are moved about a board, such as chess or checkers. The incoming event is sent as a message from the window object to the board widget (message 1). The board widget uses its layout to map mouse (x, y) coordinates onto a particular square to which the user is moving the currently selected piece, and forwards a message to that square (1.1). The square sends a message to a rules object, which checks the validity of the user’s move (1.1.1), and if the move is legal, the square sends a message to the game piece, effectively telling it to move itself (1.1.2). The game piece sends an “erase” message to the square where it was formerly located (1.1.2.1) before changing its link to refer to the square to which it is moving.

![Collaboration Diagram](image)

Figure 13-4: A Collaboration Diagram

There are a couple more annotations worth noting in Figure 13-4. The links between window, board widget, and square widget are identified as aggregations since they denote
geometric containment; this information is redundant with the class diagram, but is given to explain how the objects are linked to allow message transmission. The connections between the square widget and the rules and game piece objects are marked as \texttt{<<global>>} to indicate that the square widget obtains references to these objects from global variables. The link between the game piece and the square widget in which it is located is a regular association and does not require further annotation. Besides \texttt{<<global>>} you can annotate a link as a \texttt{<<parameter>>} or \texttt{<<local>>} to indicate other non-association references through which messages are transmitted.

13.4 Summary

This chapter introduced four UML diagram types that are useful in modeling dynamic aspects of a program’s behavior. To learn more about these techniques and others, consult a primary UML resource, such as \textit{The Unified Modeling Language User Guide}, by Grady Booch, James Rumbaugh, and Ivar Jacobson.

No one technique is a complete solution, but some combination of use cases, statecharts, and collaboration diagrams will allow you to sufficiently model most applications. Use cases are particularly valuable for describing tasks from the point of view of the application domain and human user. Statecharts are good for modeling event-based systems such as user interfaces or distributed network applications. Collaboration diagrams describe interactions between objects that allow you to model the big picture in a complex system.

In terms of primacy and chronological order, for most applications you should start with use cases and try to develop them completely. For those use cases that seem complex, or for which the conventional use case description seems inadequate, you can then bring in statecharts or collaboration diagrams to assist in completing an understandable design.

Class diagrams are the backbone of a detailed object oriented design. They can be developed by extracting details from the other kinds of diagrams, and should reflect programmers’ understanding of the application domain for which the software is being written.
Part III

Example Applications
Chapter 14

CGI Scripts

CGI scripts are programs that read input forms and generate dynamic HTML content for the World Wide Web. CGI programs are often written in scripting languages, but they can be written in any language, such as C. Unicon is ideal for writing CGI scripts, since it has extraordinary support for string processing. In this chapter you will learn how to

- Construct programs whose input comes from a web server.
- Process user input obtained from fields in HTML forms
- Generate HTML output from your Icon programs

14.1 Introduction to CGI

The Common Gateway Interface, or CGI, defines the means by which Web servers interact with external programs that assist in processing Web input and output. CGI scripts are programs that are invoked by a Web server to process input data from a user, or provide users with pages of dynamically generated content, as opposed to static content found in HTML files. The primary reference documentation on CGI is available on the Web from the National Center for Supercomputer Applications (NCSA) at http://hoohoo.ncsa.uiuc.edu/cgi/. If you need a gentler treatment than the official reference, *The CGI Book*, by Bill Weinman, is a good book on CGI. Although other methods for writing web applications on the server have been developed, CGI is the most general, portable method and is likely to remain in wide use for some time.

This chapter describes *cgi.icn*, a library of procedures for writing CGI scripts. The *cgi.icn* library consists of a number of procedures to simplify CGI input processing and especially the generation of HTML-tagged output from various data structures. The *cgi.icn* reference documentation can be found in Appendix B, which describes many important modules in the Icon Program Library.

Note
To use \texttt{cgi.icn}, place the statement \texttt{link\ cgi} at the top of your program.

CGI programs use the hypertext markup language HTML as their output format for communicating with the user through a Web browser. Consequently, this chapter assumes you can cope with HTML, which is beyond the scope of this book. HTML is an ASCII format that mixes plain text with tags consisting of names enclosed in angle brackets such as \texttt{<HTML>}. HTML defines many tags. A few common tags will be defined where they occur in the examples. Most tags occur in pairs that mark the beginning and end of some structure in the document. End tags have a slash character preceding the name, as in \texttt{</FONT>}. More details on HTML are available from the World Wide Web Consortium at http://www.w3.org/MarkUp/.

**Organization of a CGI script**

CGI programs are very simple. They process input data supplied by the Web browser that invoked the script (if any), and then write a new Web page, in HTML, to their standard output. When you use \texttt{cgi.icn} the input-processing phase is automatically completed before control is passed to your program, which is organized around the HTML code that you generate in response to the user. In fact, \texttt{cgi.icn} includes a \texttt{main()} procedure that processes the input and writes HTML header and tail information around your program’s output. For this reason, when you use \texttt{cgi.icn}, you must call your main procedure \texttt{cgimain()}.

**Processing input**

The HTTP protocol includes two ways to invoke a CGI program, with different methods of supplying user input, either from the standard input or from a \texttt{QUERY_STRING} environment variable. In either case, the input is organized as a set of fields that were given names in the HTML code from which the CGI program was invoked. For example, an HTML form might include a tag such as:

\begin{verbatim}
<INPUT TYPE = "text" NAME = "PHONE" SIZE=15>
\end{verbatim}

which allows input of a string of length up to 15 characters into a field named \texttt{PHONE}.

After the CGI library processes the input, it provides applications with the various fields from the input form in a single table, which is a global variable named \texttt{cgi}. The keys of this table are exactly the names given in the HTML \texttt{INPUT} tags. The values accessed from the keys are the string values supplied by the user. For example, to access the \texttt{PHONE} field from the above example, the application could write

\begin{verbatim}
cgi["PHONE"]
\end{verbatim}
14.1. INTRODUCTION TO CGI

Processing output

The main task of the CGI program is to write an HTML page to its standard output, and for this task \texttt{cgi.icn} provides a host of procedures. Typically these procedures convert a structure value into a string, wrapped with an appropriate HTML tag to format it properly. A typical example is the library procedure \texttt{cgiSelect(name,values)}, which writes an HTML \texttt{SELECT} tag for a field named name. The \texttt{SELECT} tag creates a list of radio buttons on an HTML form whose labels are given by a list of strings in the second parameter to \texttt{cgiSelect()}. A programmer might write

\[
\texttt{cgiSelect("GENDER", ["female", "male"])}
\]

to generate the HTML

\[
<\texttt{SELECT NAME="GENDER">}
<\texttt{OPTION SELECTED}>female
<\texttt{OPTION}>male
</\texttt{SELECT}>
\]

Common CGI environment variables

The official CGI definition includes a set of standard environment variables that are set by the Web server as a method of passing information to the CGI script. Programmers access these environment variables using \texttt{getenv()}, as in

\[
\texttt{getenv("REMOTE_HOST")}
\]

Table 14-1 presents a summary of the CGI environment variables as a convenience so that this book can serve as a stand-alone reference for writing most CGI scripts. For a complete listing of all the environment variables supported by CGI go to \url{http://hoohoo.ncsa.uiuc.edu/cgi/env.html} on the Internet.

Table 14-1

<table>
<thead>
<tr>
<th>CGI Environment Variables</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENT_LENGTH</td>
<td>The length of the ASCII string provided by method=&quot;POST&quot;.</td>
</tr>
<tr>
<td>HTTP_USER_AGENT</td>
<td>The user’s browser software and proxy gateway, if any. The format is name/version, but varies wildly.</td>
</tr>
<tr>
<td>QUERY_STRING</td>
<td>The information submitted through the form, which follows the ? in the URL when using method=&quot;GET&quot;. QUERY_STRING data is parsed and inserted into a table stored in the global variable cgi, so cgi.icn scripts do not generally consult this environment variable.</td>
</tr>
<tr>
<td>REMOTE_ADDR</td>
<td>The IP address of the client machine.</td>
</tr>
<tr>
<td>REMOTE_HOST</td>
<td>The hostname of the client machine. Defaults to IP held by REMOTE_ADDR.</td>
</tr>
<tr>
<td>REQUEST_METHOD</td>
<td>The method (GET or POST) used to invoke the CGI script.</td>
</tr>
<tr>
<td>SERVER_NAME</td>
<td>The server’s hostname. It defaults to the IP address.</td>
</tr>
<tr>
<td>SERVER_SOFTWARE</td>
<td>The Web server that invoked the CGI script. The format is name/version.</td>
</tr>
</tbody>
</table>

### 14.2 The CGI Execution Environment

CGI scripts do not execute as stand-alone programs and aren’t launched from a command line; a Web server executes them. The details of this are necessarily dependent on the operating system and Web server combination in use. The following examples are based on a typical UNIX Apache server installation in which users’ HTML files are located under $HOME/public_html. Check with your system administrator or Web server documentation for the specific filenames, directories, and permissions required to execute scripts from your Web server. Some web servers do not allow scripts at all, and most others run scripts with a special userid in a limited/protected file system where absolute pathnames are different from how you see them.

Under Apache, you need a directory under $HOME/public_html named cgi-bin. Both $HOME/public_html and its cgi-bin subdirectory should have "group" and "other" permissions set to allow reading and executing for the Web server to run the programs you place there. Do not give anyone but yourself write permissions! The following commands set things up on a typical Apache system. The percent sign (%) is not part of the command; it is the UNIX shell prompt. The period in the final command is part of the command and refers to the current working directory.

```bash
% mkdir $HOME/public_html
% cd $HOME/public_html
% mkdir cgi-bin
% chmod go+rx . cgi-bin
```
The next two example files will allow you to verify that your directories and permissions are correct for your Web server. Despite all the attempts to make the world’s web servers secure, the only security you can count on is your own. From security expert David A. Gamey we have the following tips:

- Use no data without checking for validity. Even HTTP header data can be wrong. If you expect a number, make sure the supplied data is a number.
- It is a very bad idea to make any `system()` calls (or open piped commands, etc.) from scripts. If you absolutely have to run something external, construct command strings yourself or fully parse user data being used to check for command separators and hidden commands, etc.
- Don’t rely on looking for known bad characters; restrict input to known good characters. Use known good values such as those selected from list boxes
- Sensitive data should be sent using POST, not GET.
- Check for and prevent file system attacks, such as paths including .. in them.
- Log everything received by your script, so you can tell when attacks occur.

14.3 An Example HTML Form

CGI scripts are typically invoked from HTML pages. When you view the following example page in your browser, it should look something like the one shown in Figure 14-1. For this test, create an HTML form `$HOME/public_html/simple.html` containing Listing 14-1. When you have a CGI script compiled and ready to run, you can edit the URL in this file to point at your CGI program, the `simple.cgi` executable.
CHAPTER 14. CGI SCRIPTS

Figure 14-1: An HTML Form Example

Listing 14-1 An HTML form

```html
<HTML>
<HEAD><title>An HTML Form Example</title></HEAD>
<BODY>
<h1>A cgi.icn Demonstration</h1>
<form method="GET" action="http://www.cs.uidaho.edu/~jeffery/cgi-bin/simple.cgi">
1. Name: <input type="text" name="name" size=25> <p>
2. Age: <input type="text" name="age" size=3> Years <p>
3. Quest: <input type="checkbox" name="fame">Fame <input type="checkbox" name="fortune">Fortune <input type="checkbox" name="grail">Grail <p>
4. Favorite Color: <select name="color">
<option>Red <br>
<option>Green <br>
<option>Blue <br>
<option selected>Don't Know (Aaagh!) <br>
</select><p>
Comments:<br>
</form>
</BODY>
</HTML>
```
14.4 An Example CGI Script: Echoing the User’s Input

The following script, named `simple.cgi` might be invoked from the FORM tag above. The `simple.cgi` script is produced from an Unicon source file, `simple.icn`, that you can copy from the book web site ([http://unicon.org/book/](http://unicon.org/book/)). This program needs to be compiled with the command

```
unicon -o simple.cgi simple.icn
```

Many Web servers are configured so that CGI scripts must end with the extension `.cgi`. Check with your system administrator about CGI naming conventions if the `.cgi` extension does not work for you. In addition to the web server being configured to allow user invocation, unless you use compiler option `-B` to bundle the virtual machine into your executable, your program must be able to find and execute the virtual machine from whatever user id CGI’s are executed.

The program reads the form input specified in `simple.html`, and writes it back out. All `cgiEcho()` is doing in this case is adding an HTML newline tag after each call. If you look it up in Appendix B, you will find that it will copy its arguments to both the HTML output and a log file if given a file as its first argument.

```unicon
link cgi
procedure cgimain()
    cgiEcho("Hello, ", cgi["name"], ",!")
    cgiEcho("Are you really ", cgi["age"], ", years old?")
    cgiEcho("You seek: ", cgi["fame"] == = "on" & "fame")
    cgiEcho("You seek: ", cgi["fortune"] == = "on" & "fortune")
    cgiEcho("You seek: ", cgi["grail"] == = "on" & "grail")
    cgiEcho("Your favorite color is: ", cgi["color"])
    cgiEcho("Your comments: ", cgi["comments"])
end
```

Generating an output page that rehashes the user’s input is a good test of your HTML form before you deploy it with a CGI script that actually does something. In some cases, it is also helpful in allowing the user to recheck their submitted input and confirm or cancel before acting on it.
14.5 Debugging CGI Programs

CGI programs can be a pain to debug. You may have to debug your CGI execution environment, before you can even start debugging your CGI program itself. If your CGI script returns an "internal server error", or no output at all, you may have file permissions wrong, or the CGI script may not be able to find the Unicon virtual machine in order to run the program. Some web servers execute CGI scripts under a special userid such as "web", others will run them under your user id. Some web servers run CGI scripts under a protected file system where the root directory "/" is not the same as the root directory visible to your user account, so the path to iconx that you normally use may be invalid in your CGI program. CGI scripts may have a very limited PATH for security reasons, not the PATH you set for your user account. Your best bet is probably to use the -B Unicon compiler option to bundle the Unicon interpreter into your executable file; alternatively you can probably copy the virtual machine "iconx" into your cgi-bin directory.

Debugging your CGI program itself may require special tricks. Because your CGI program is executed by a web server, its standard error output may not be visible to you. You can try to redirect error output to standard out, but your error output may not be readable unless it is converted into HTML (say, by adding \(<\text{BR}>\) at each newline). One way to accomplish this is to write two programs: one that performs the primary task, and a second program that calls the first one, catches any error messages, and converts any plain text output to HTML.

14.6 Appform: An Online Scholarship Application

The next example, appform.icn, is a CGI script for an on-line scholarship application that was used at a university. Its structure is similar to the previous example, with a twist: the user input is printed for the convenience of the scholarship administrators. As a backup, the CGI script also e-mails the application to the scholarship administrator. This is useful if the print job fails for some reason. The program is a single cgimain() procedure, which starts by processing each of the user input fields. The program then opens a temporary file with a .txt extension, and writes a nicely formatted document containing the user's scholarship application information.

The code for Appform is shown in Listing 14-2. To run it you must adapt it to your environment. As written, it prints to lpr, and sends mail to jeffery@cs.uidaho.edu. When running a CGI script it is important to realize you will run in a different directory, and with different user id and PATH environment, than your regular account. The program runs with whatever user id and permissions the system administrator assigns the Web server process. For example, its root (/) directory may not be at the root of your regular filesystem, so absolute paths may not work.

Listing 14-2
An online application form

# File: appform.icn
# Subject: CGI program to process scholarship applications
# Author: Clinton Jeffery
# Date: July 11, 2002

This program processes a bunch of input fields defined in an on-line scholarship application at http://unicon.org/book/appform.html and from them, # generates a text file, prints and e-mails it to the scholarship coordinator.

link cgi, io
$define ADMIN "jeffery@cs.uidaho.edu"
procedure cgimain()
  fname := tempname("appform", ".txt", "/tmp")
  f := open(fname, "w") | stop("can't open ", fname)
  write("Generating typeset copy of application form...")
  write(f,"Scholarship Program Application\n")
  write(f, "Name: ", cgi["NAME"], " Phone: ", cgi["PHONE"])
  write(f, "Address: ", cgi["ADDRESS1"], " Social Sec. Number: ", cgi["SOC"])
  write(f,cgi["ADDRESS2"]," Gender (M/F): ",cgi["GENDER"], "")
  write(f,"Semester hours completed: ",cgi["CREDITS"])
  write(f,"College GPA: Overall ",cgi["GPA"])
  write(f,"Present Employer: ",cgi["EMPLOYER"])
  write(f,"Position: ",cgi["POSITION"]," Hours/week: ",cgi["HOURS"])
  write(f,"Educational Background")
  write(f,"High School: name, year, GPA, graduated?")
  write(f,cgi["HIGH1"], "",cgi["HIGH2"])
  write(f,"For each college, list name, dates attended, credit hours, ")
  write(f,"degrees awarded",cgi["COLL1"], ",",cgi["COLL2"], ",")
  write(f,"Academic honors, scholarships, and fellowships")
  write(f,cgi["HONOR1"], ",",cgi["HONOR2"], ",")
  write(f," Extracurricular interests: ",cgi["EXTRA1"], ",",cgi["EXTRA2"])
  write(f,"Professional organizations: ",cgi["ORGS1"], ",",cgi["ORGS2"])
  write(f,"Research interests: ")
  write(f,cgi["RESEARCH1"], ",",cgi["RESEARCH2"])
  write(f,"Name(s) of at least one person you have asked to")
  write(f,"write an academic reference letter.")
  write(f,"Name Address Relationship")
  write(f,cgi["REF1"], ",",cgi["REFADD1"], ",",cgi["REFREL1"])
  write(f,cgi["REF2"], ",",cgi["REFADD2"], ",",cgi["REFREL2"])
  write(f,"I certify that information provided on this")
  write(f,"application is correct and complete to my knowledge.")
Summary

Writing CGI scripts in Unicon is easy. The input fields are handed to you elegantly in a
global variable, and library functions allow you to write terse code that generates correct
HTML output. The only thing certain about the fast-changing Internet standards is that
they will get continually more complex at a rapid pace. CGI scripting is no substitute for
JavaScript, XML, or any newer buzzword that may be hot this week. But it is a lasting,
multiplatform standard for how to run a program on a Web server from a browser, and it
may be the simplest and best solution for many Internet applications for some time.
Chapter 15

System and Administration Tools

In an open computing environment, users build their own tools to extend the capabilities provided by the system. Unicon is an excellent language for programmers who wish to control and extend their own system. This chapter presents techniques used to write several system utilities of interest to general users as well as system administrators. Best of all, many of these utilities work across multiple platforms, thanks to Unicon’s high degree of system portability. You will see examples of

- Traversing and examining directories and their contents.
- Finding duplicate files.
- Implementing a quota system for disk usage.
- Doing your own custom backups.
- Capturing the results of a command-line session in a file.

15.1 Searching for Files

To begin, consider a simple problem: that of finding a file whose name matches a specified pattern. Regular expressions are commonly used to describe the patterns to match, so you may want to link in the regular expression library. Here is the start of a file-search application.

```unicon
#
# search.icn
#
# Search for files whose (entire) names match a pattern given
# as a regular expression
#
# Usage: ./search <pattern> [dirs]
```
The application starts by processing the command-line arguments. There must be at least one argument: the pattern to search for. Arguments following that one are directories to search. If no directories are specified, you can use the current working directory. The procedure `findfile()` performs the actual task of searching for the files:

```plaintext
procedure main(args)
    (*args > 0) | stop("Usage: search <pattern> [directories]")
    pattern := pop(args)
    if *args = 0 then findfile(".", pattern)
    else
        every dir := !args do findfile(dir, pattern)
    exit(0)
end
```

The search algorithm is a depth-first search. In each directory, check the type of each file. If you find a directory, make a recursive call to `findfile()`. But before you do that, check to see if the name of the file matches the pattern.

For efficiency, since the program uses the same regular expression for all searches, you can compile the pattern into a static variable called `pat`. The regular expression library allows you to perform this compilation once, the first time that `findfile()` is called, and then reuse it in subsequent calls.

```plaintext
procedure findfile(dir, pattern)
    local d, f, s
    static pat
    initial {
        pat := RePat(pattern) | stop("Invalid pattern ", image(pattern))
    }
    d := open(dir) | {
        write(&errout, "Couldn’t access ", dir, " ", sys_errstr(&errno))
        return
    }
```

While you read the names of the files in the directory, be sure to not go into the special entries "." and ".." that represent the current directory and the parent directory, respectively. Except for these names, the directory hierarchy is a tree so you don't need to check for cycles. Some file systems support the concept of links, described in Chapter 5; links can introduce cycles, so the code below recursively calls itself only on regular directories, not on links.
while name := read(d) do {
    if name == ("." | ".") then next
    f := dir || "/" || name
    s := stat(f) | {
        write(&errout, "Couldn't stat ", f,
        " ", sys_errstr(&errno))
        next
    }
}

Here is the check of the file name against the pattern:

    name ? if tab(ReMatch(pat)) & pos(0) then write(f)

Note

Regular expressions do not use the same notation as file-matching wildcards used on
the command line. The regular expression notation used by RePat() and ReMatch() is
given in the documentation for the regexp module in Appendix B.

Finally, if f is the name of a directory, make the recursive call. Note that since the
pattern has already been compiled and stored in a static variable, you don't need to pass
it in as a parameter for the recursive call.

    if s.mode[1] == "d" then findfile(f)
}
close(d)
close(d)
end

This is a very simple demonstration of some systems programming techniques for Uni-
con. You will see this sort of depth-first traversal of the file system again, in the section on
file system backups later in this chapter.

15.2 Finding Duplicate Files

An interesting variation on the previous program is to find files whose contents are identical.
This is valuable for those of us who make many copies of files in various subdirectories over
time, and then forget which ones are changed. Since this task deals with lots of files, there
are some things to think about. Reading a file is an expensive operation, so you should try
to minimize the files you read. Since you can find the length of a file without reading it,
you can use that to perform the first cut: you won't need to compare files of different sizes.
The first step, then, is to solve the simpler problem of finding files that have the same size.

The previous program example shows how to traverse the directory structure. For the
lengths you can use a table - with each possible length, store the names of the files of that
length. Since there are lots of files, try to be smart about what you store in the table. The
natural structure is a list. This leads to the following code:
procedure scan(dir)
    f := open(dir) | {
        write(&errout, "Couldn't access ", dir, ", ", sys_errstr(&errno))
        return
    } while name := read(f) do {
        filename := dir || "/" || name
        r := stat(filename)
        case r.mode[1] of {
            "-" : {
                /lengths[r.size] := list()
                push(lengths[r.size], filename)
            }
            "d" : name == ("." | "..") | scan(filename)
        }
    }
    close(f)
end

The main program scans all the directories, and for each list, compares all the files in it to each other. The global table named lengths maps lengths to filenames.

global lengths
procedure main()
    lengths := table()
    scan("/.")
    every l := !lengths do {
        if *l = 1 then next
        find_dups(l)
    }
end

For example, if in a directory there are files A, B, C, and D with lengths of 1, 2, 5, and 2 bytes, respectively, the lengths table will contain the following:

lengths[1] === [ "A" ]
lengths[2] === [ "B", "D" ]
lengths[5] === [ "C" ]

If a list only has one element, there is no reason to call the function to compare all the elements together.

All of this makes sense, but in many cases there will be only one file that has a certain size. Creating a list for each file size is a small waste of space. What if, for the first entry, you only store the name of the file in the table? Then if you get a second file of the same size, you can convert the table entry to store a list. That is, in the above example you could have
Now for most of the files, the program is only storing a string, and it creates a list only where it needs one. You can say that the table is heterogeneous if you want to get technical about how its elements are a mixture of strings and lists. With this change, the main procedure becomes:

```plaintext
global lengths
procedure main()
    lengths := table()
    scan("/")
    every l := !lengths do {
        if type(l) == "string" then next
        find_dups(l)
    }
end
```

Instead of checking to see if the list has only one element, the code checks to see if the value from the table is a string, and ignores those entries.

The scan procedure has to do a little more work. Instead of initializing the value to a list, you can use the name of the current file; if the value already in the table is a string, create a list and add both the name from the table and the name of the current file to the list. If the value in the table is a list already, then you can just add the current filename to it.

```plaintext
while name := read(f) do {
    filename := dir || "/" || name
    r := stat(filename)
    case r.mode[1] of {
        ":
            case type(lengths[r.size]) of {
                "null" : lengths[r.size] := filename
                "string" : {
                    lengths[r.size] := [lengths[r.size]]
                    push(lengths[r.size], filename)
                }
                "list" : push(lengths[r.size], filename)
            }
            "d" : name == ("." | ".") | scan(filename)
    }
}
To compare two files together, you will of course have to read both files. One way to do it would be to read in the files to memory and compare them, but that would take a lot of space. Most files even when they have the same size will probably be different; you only need to read the files until you find a difference. At the same time, you shouldn’t read the files one byte at a time, since the I/O system is optimized to read larger chunks at a time. The chunk size to use will depend on the exact configuration of the computer the program is running on, that is, the speed of the file reads compared to the CPU speed and the available RAM storage.

The actual comparison is simple: keep reading chunks of both files, failing if you find a difference and succeeding if you reach the ends of the files without finding any.

```
procedure compare(file1, file2)
    static maxline
    initial maxline := 1000
    f1 := open(file1) | fail
    f2 := open(file2) | { close(f1); fail }
    while l1 := reads(f1, maxline) do {
        l2 := reads(f2, maxline)
        if l1 ˜== l2 then {
            every close(f1 | f2)
            fail
        }
    }
    every close(f1 | f2)
    return
end
```

One technique that is sometimes used for comparing long strings and so forth is hashing. To use hashing, you define a function that computes an integer from the string. If the hash values of two strings are different, you know that they cannot be the same. However hash values being equal doesn’t necessarily imply that the strings are equal, so in the worst case scenario you still have to compare the two strings. If you aren’t familiar with hashing, we encourage you to consult a book on algorithms to learn more about this technique and think about how the program may be further improved with it. One example hash function used internally by Unicon is equivalent to:

```
procedure hash(s)
    local i := 0
    every i +:= ord(s[1 to min(*s, 10)]) do
        i *:= 37
        i +:= *s
    return i
end
```
To complete the application, one piece remains: given a list of filenames, you need to go through them all and make all the possible comparisons. This can be done with a simple loop, calling \texttt{compare()} to perform the actual comparisons.

\begin{verbatim}
procedure find_dups(l)
    every i := 1 to *l do {
        f1 := l[i]
        every j := i+1 to *l do {
            if compare(f1, l[j]) then write(f1, " == ", l[j])
        }
    }
end
\end{verbatim}

This is relatively inefficient: for a list of \(n\) files, it performs \(n(n-1)/2\) comparisons and reads each file \(n-1\) times. In certain situations it is possible to get by with not doing so many comparisons; we leave this as an exercise to the reader.

\textbf{Tip}

Clever use of hashing might let you get away with reading each file just once.

Listing 14-1 shows the complete program, with added comments, and also some error checking. If any directory or file open fails, it prints a message and proceeds with the processing.

Listing 14-1 A program for finding duplicate files.

\begin{verbatim}
#
# duplicate.icn
#
# Find files in the filesystem that are identical

global lengths
procedure main()
    lengths := table()
    # On some systems, a leading "/" in a filename may have
    # a different meaning, so we use "." instead of just "/"
    scan("./")
    every l := !lengths do {
        if type(l) == "string" then next
        find_dups(l)
    }
    exit(0)
end

# Scan all the directories and add files to the length map -
# the global table "lengths"
\end{verbatim}
procedure scan(dir)
    f := open(dir) | {
        write(&errout, "Couldn't open ", dir, "; ", sys_errstr(&errno))
        fail
    }
    while name := read(f) do {
        filename := dir || "/" || name
        r := stat(filename) | {
            write(&errout, "Couldn't stat ", filename, "; ", sys_errstr(&errno))
            next
        }
        # A small optimisation: there are probably quite a few
        # zero-length files on the system; we ignore them all.
        r.size > 0 | next
        case r.mode[1] of {
            ":
                # ordinary file
                case type(lengths[r.size]) of {
                    # if null, it's the first time; just store filename
                    "null" : lengths[r.size] := filename
                    # if string, one element already exists; create a list and make
                    # sure we add both the old filename and the new one.
                    "string" : {
                        lengths[r.size] := [lengths[r.size]]
                        push(lengths[r.size], filename)
                    }
                    # There's already a list; just add the filename
                    "list" : push(lengths[r.size], filename)
                }
            "d"
                # A directory. Make sure to not scan . or ..
                name == ("." | ".") | scan(filename)
        }
    }
    close(f)
    return ""
end

# Given a list of filenames, compare the contents of each with every other.
procedure find_dups(l)
    # This is O(n^2)
15.3. USER FILE QUOTAS

```plaintext
every i := 1 to *l do {
    f1 := l[i]
    every j := i+1 to *l do {
        if compare(f1, l[j]) then write(f1, " == ", l[j])
    }
}
end

# Compare two files; by reading in 1000 byte chunks. This value may need
# to be adjusted depending on I/O speed compared to CPU speed and memory.
procedure compare(file1, file2)
    static maxline
    initial maxline := 1000

    # are f1 and f2 identical?
    f1 := open(file1) | {
        write(&errout, "Couldn't open ", file1, "; ", sys_errstr(&errno))
        fail
    }
    f2 := open(file2) | {
        close(f1)
        write(&errout, "Couldn't open ", file2, "; ", sys_errstr(&errno))
        fail
    }
    while l1 := reads(f1, maxline) do {
        l2 := reads(f2, maxline) |
        # The files are supposed to be the same size! How could
        # we read from one but not the other?
        stop("Error reading ", file2)
        if l1 ˜== l2 then {
            every close(f1 | f2)
            fail
        }
    }
    every close(f1 | f2)
    return
end
```

15.3 User File Quotas

Many computing platforms offer a filesystem quota facility, where each user has only so
much of the disk to store files. The system does not allow files to grow once the limit
has been reached. However, many systems don’t have this facility, and on other systems
the available quota mechanism is not enabled because the user might have an urgent and immediate need to exceed his or her quota for a short time.

For these uses this section presents an alternate filesystem quota method. The quota for each user is stored in a file, and at regular intervals (perhaps overnight) the system examines the disk usage of each user. If it is above the quota, a message is sent. A summary message is also sent to the administrator so that any user that is over quota for more than a few days will be noticed.

First, declare a few global variables that will hold the strings that are to be used in the messages that are sent:

```unicon
  global complaint_part1, complaint_part2, summary_header
```

```unicon
  procedure main(args)
    # Read database of users; get disk usage for each user and
    # check against his/her quota
    db := "/usr/lib/quotas/userdb"
    administrator := "root"
    init_strings()
```

Calculating disk usage

The next step is to read the database of users, and for every directory, calculate the disk usage. On UNIX systems you can just run `du` (the UNIX tool that measures disk usage) and read its output from a pipe to obtain this figure. The `-s` option tells `du` to print a summary of disk usage; it prints the usage and the name of the directory, separated by a tab character. If your platform doesn’t have a `du` command, how hard is it to write one in Unicon? The `du` command is straightforward to write, using the techniques presented in the previous two programming examples.

```unicon
  procedure du(dir)
    local s := stat(dir), d, sum := 0

    # If it’s not a directory, just return its size
    if s.mode[1] == “d” then
      return s.size

    # Otherwise, find the usage of each entry and add
    d := open(dir) | fail
    while filename := read(d) do sum +:= du(filename)
    close(d)
    return sum
  end
```

Using this procedure, the program fills in the table of usages.
15.3. **USER FILE QUOTAS**

\[
L := \text{read\_db}(db) \\
\text{owners} := L[1] \\
\text{quotas} := L[2] \\
\text{daysover} := L[3] \\
\text{usages} := \text{table}(0) \\
\text{over} := \text{table()} \\
every \text{dir} := \text{key(owners)} \text{ do } \{ \\
\text{usages}[\text{dir}] := \text{du(\text{dir})} \\
\text{user} := \text{owners}[\text{dir}] \\
\}
\]

If the usage reported is greater than the quota, increment the "days over" field. Save the results and send all the email later; this will allow us to only send one message to a user that owns more than one directory.

\[
\text{if } \text{usages}[\text{dir}] > \text{quotas}[\text{dir}] \text{ then } \{ \\
/\text{over}[\text{user}] := [] \\
\text{daysover}[\text{dir}] := 1 \\
\text{l} := [\text{dir}, \text{usages}[\text{dir}], \text{quotas}[\text{dir}], \text{daysover}[\text{dir}]] \\
\text{push(over}[\text{user}], \text{l}) \\
\} \\
\text{else daysover}[\text{dir}] := 0 \\
\}
\]

Finally, the program saves the database and sends a summary of the over-quota directories to the administrator.

\[
\text{write\_db}(db, \text{owners}, \text{quotas}, \text{daysover}) \\
\text{send\_summary(administrator, owners, quotas, daysover, usages)} \\
\]

**Sending mail messages**

Two procedures in the quota program send mail messages as their primary task. This is done in many older system administration scripts by executing an external mail client using the `system()` function. Calling `system()` is a potential portability problem and security hole in many applications. The quota program uses Unicon’s messaging facilities to send mail, avoiding both of these problems.

Procedure `complain()` sends a message to the user notifying him/her that certain directories are over quota. The entry for each user is a list, each member of which is a record of an over-quota directory, stored as a list. This list has the directory name, the usage, the quota and the number of days it has been over quota.
procedure complain(user, L)
    msg := "Dear " || user || complaint_part1
    every l := !L do {
    }
    msg ||:= complaint_part2
    m := open("mailto:"|| user, "m", "Subject: Over Quota")
    write(m, msg)
    close(m)
end

Procedure send_summary() sends mail to the administrator summarizing all over-quota directories. It uses the function key() to generate the indexes for tables usages, quotas, owners, and daysover, which are maintained in parallel. This use of key() is quite common. It might be possible to combine all these parallel tables into one big table and eliminate the need to call key().

procedure send_summary(admin, owners, quotas, daysover, usages)
    m := open("mailto:"|| admin, "m", "Subject: Quota Summary")
    write(m, summary_header)
    every dir := key(owners) do
        if usages[dir] > quotas[dir] then {
            writes(m, dir, "\t", owners[dir], "\t", usages[dir]|"/"||quotas[dir], "\t",
            daysover[dir])
            # Flag anything over quota more than 5 days
            if daysover[dir] > 5 then writes(m, "****")
            write(m)
        }
    close(m)
end

The quota database

The database is stored as a plain text file so that the system administrator can easily make changes to it. It has four fields, separated by white space (spaces or tabs): a directory, the owner of the directory, the quota, and the number of days the directory has been over quota. Procedure read_db() reads in the database. Blank lines or lines starting with '#' are ignored.

procedure read_db(db)
    owners := table()
    quotas := table(0)
    daysover := table(0)
    dbf := open(db) | stop("Couldn’t open ",db)
15.3. USER FILE QUOTAS

while line := read(dbf) || "$t" do
    line ? {
        tab(many("\t"))
        if pos(0) | = "#" then next
        dir := tab(upto("\t")); tab(many(" \t"))
        user := tab(upto("\t")); tab(many(" \t"))
        quota := tab(upto("\t")); tab(many(" \t"))
        days := tab(0) | ""
        # The "days" field can be absent in which case 0 is
        # assumed.
        if days == "" then days := 0
        if \owners[dir] ˜== user then
            write(&errout, "Warning: directory ", dir, " has more than one owner.")
            owners[dir] := user
            quotas[dir] +:= quota
            daysover[dir] := days
        }
    close(dbf)
    return [owners, quotas, daysover]
end

Procedure write_db() rewrites a quota database with current quota information. Notice
how the code preserves the comments and the blank lines that were present in the database
file. This is very important when dealing with human-editable files. It also writes to a
temporary file and then renames it to the correct name. This ensures that a consistent
copy of the database is always present.

procedure write_db(db, owners, quotas, daysover)
    new_db := db || ".new"
    db_old := open(db)
    db_new := open(new_db, "w") | stop("Couldn't open", new_db)
    while line := read(db_old) do {
        line ? {
            tab(many("\t"))
            if pos(0) | = "#" then {
                write(db_new, line)
            } else {
                write(db_new, line)
            }
        }
    close(db_new)
    return [owners, quotas, daysover]
Lastly, procedure init_strings() initializes global strings used for email messages. Concatenation is used to improve the readability of long strings that run across multiple lines.

```
procedure init_strings()
    complaint_part1 := "\n" ||
        "The following directories belonging to you are" || "over their quota:\n\n"
        "Directory \tUsage \tQuota \tDays Over\n"
    complaint_part2 := "\nPlease take care of it."
    summary_header := "\n" ||
        "Over-quota users\n\n"
        "Directory \tOwner \tUsage/Quota \tDays Over\n"
end
```

15.4 Capturing a Shell Command Session

Many applications including debugging and training can benefit from the ability record a transcript of a session at the computer. This capability is demonstrated by the following program, called script. The script program uses a feature of POSIX systems called the pty. This is short for pseudo-tty. It is like a bi-directional pipe with the additional property that one end of it looks exactly like a conventional tty. The program at that end can set it into "no-echo" mode and so forth, just like it can a regular terminal. This application’s portability is limited to the UNIX platforms.

The script program has only one option: if -a is used, output is appended to the transcript file instead of overwriting it. The option is used to set the second argument to open():

```
# script: capture a script of a shell session (as in BSD)
# Usage: script [-a] [filename]
# filename defaults to "typescript"
```

procedure main(L)
if L[1] == "-a" then {
    flags := "a"; pop(L)
} else flags := "w"

Now the program must find a pty to use. One method is to go down the list of pty device names in sequence until an open() succeeds; then call procedure capturesession(), to perform the actual logging. On POSIX systems the pty names are of the form /dev/ptyp-s0-a. The tty connected to the other end of the pipe then has the name /dev/ttyXY, where X and Y are the two characters from the pty’s name.

# Find a pty to use
every c1 := !"pqrs" do
    every c2 := !(&digits || "abcdef") do
        if pty := open("/dev/pty" || c1 || c2, "rw") then { # Aha!
            capturesession(fname := L[1] | "typescript", pty, c1 || c2, flags)
            stop("Script is done, file ", image(fname))
        }
    stop("Couldn’t find a pty!")
end

Note

If you do not have read-write permissions on the pseudotty device the program uses, the program will fail. If this program does not work, check the permissions on the /dev/tty* device it is trying to use.

The script program uses the system() function, executing the user’s shell with the standard input, standard output, and standard error streams all redirected to be the tty end; then it waits for input (using select()) either from the user or from the spawned program. The program turns off echoing at its end, since the spawned program will be doing the echoing. The program sends any input available from the user to the spawned shell; anything that the shell sends is echoed to the user, and also saved to the script file.

procedure capturesession(scriptfile, pty, name, flags)
    f := open(scriptfile, flags) | stop("Couldn’t open ", image(scriptfile))
    tty := open("/dev/tty" || name, "rw") | stop("Couldn’t open tty!")
    shell := getenv("SHELL") | "/bin/sh"
    system([shell, "-i"], tty, tty, tty, "nowait")

    # Parent
    close(tty)
    system("stty raw -echo")

    # Handle input
while L := select(pty, &input) do {
  if L[1] === &input then writes(pty, reads()) | break
  else if L[1] === pty then {
    writes(f, inp := reads(pty)) | break
    writes(inp)
  }
}

When script gets an EOF on either stream, it quits processing and closes the file, after resetting the parameters of the input to turn echoing back on.

(&errno = 0) | write(&errout, "Unexpected error: ", sys_errstr(&errno))
  system("stty cooked echo")
  close(f)
end

15.5 Filesystem Backups

By now you have probably been told a few thousand times that regular backups of your files are a good idea. The problem arises when you are dealing with a system with a large amount of file storage, like modern multi-user systems. For these systems, incremental backups are used. Incremental backups exploit the fact that a very large number of files on the system change rarely, if ever. There is no need to save them all to the backup medium every time. Instead, each backup notes the last time that a backup was performed, and only saves files that have been modified since then.

Each backup is given a number; the files in backup n were modified later than the last backup of a lower number. A Level 0 backup saves all the files on the system.

This section presents an incremental backup utility called backup. The backup utility saves all the files in a directory specified with the -o flag. Typically this will be the external backup storage, like a floppy disk (for small backups) or a Zip disk. The program re-creates the directory structure on the external device so that files may easily be recovered. The disadvantage of this strategy is that it does not compress the whole archive together, and therefore requires more storage than is strictly necessary. On the positive side, this approach avoids the fragility of compressed archives, in which the loss of even a small amount of data can render the whole archive unreadable.

Note

This program only saves to media that has a directory structure on which regular files may be written, such as jump drives. It does not work on backup devices such as tape drives that require media to be written in a proprietary format.

Another feature of this backup program is that certain directories can be automatically excluded from the backup, such as temporary directories like /tmp on UNIX systems. One
directory that must be excluded is the output device itself, or you will find that a very large amount of storage is needed! One of the best parts about backup is that you can modify this program to suit your needs: file compression, error recovery, support for multiple discs, or anything else that you require.

```plaintext
# backup.icn - incremental filesystem backups
#
# Usage:
# ./backup [-nlevel] [-ooutput] [dir]
#
# Save all files that have changed since the last backup of higher level
# (a level 0 backup is the highest level and saves all files; it is the
# default). The files are all saved to the directory "output", which is
# probably a mounted backup device like a flash drive.
#
# Example:
# backup -n3 -o/mnt/zip /home/bob

link options

global dbase, exclude, levels
global output, last

procedure main(args)
    dbase := "/var/run/backups.db"
    exclude := ["/mnt", "/tmp", "/dev", "/proc"]

    # Process arguments
    opt := options(args, ":-n+-o:")
    level := integer(opt["n"] | 0)
    output := opt["o"]
    dir := args[1] | "/

    \output | stop("An output directory (option -o) must be specified!")
    if level < 0 | level > 9 then stop("Only levels 0..9 can be used.")

    # Get the time of the previous lower-numbered backup
    last := get_time(level)
    # Now look for files newer than "last"
    traverse(dir)

    # Write the database
    save_time(level)
```
**CHAPTER 15. SYSTEM AND ADMINISTRATION TOOLS**

Procedure **traverse()** is the interesting part of the program. It recursively descends the filesystem hierarchy, saving all the files it finds that are newer than the last backup. When a recent plain file is found, the procedure **copy_file()** is called.

```plaintext
procedure traverse(dir)
    # Skip excluded directories
    if dir == !exclude then return

    # Read all the files; for any non-special files, copy them
    # over to the output dir, creating directories as necessary
    d := open(dir) | {
        write(&errout, "Couldn't stat ", dir, " ", sys_errstr(&errno))
        return
    } if dir[-1] ˜== "/" then dir ||:= "/
    while name := read(d) do {
        if name == ("." | ".") then next
        s := stat(dir || name) | {
            write(&errout, "Couldn't stat ", dir||name, " ", sys_errstr(&errno))
            next
        } if s.mode[1] == "d" then traverse(dir || name)
        else {
            # Only save plain files
            if s.mode[1] == "-" & s.cctime > last then copy_file(dir, name)
        }
    }
end
```

To copy a file, you must first ensure that its parent directory exists. If it doesn’t, the parent directory is created; then the file itself is copied. For efficiency, **backup** uses the system program (**cp** on UNIX) to copy the file. When directories are created, **backup** copies the owner and mode from the directory being backed up.

Note
This program must be run with administrator privileges for it to be able to read all the files on the system and also to be able to set the owner and mode.

```plaintext
procedure copy_file(dir, name)
    # First, make sure the directory exists
    mkdir_p(output, dir)
    system("cp " || dir || "/" || name || " " || output || "/" || dir)
end
```
procedure mkdir_p(prefix, dir)
    # The name is supposed to be reminiscent of "mkdir -p"
    # Start at the first component and keep going down it,
    # copying mode and owner.
    dir ||= "/"
    d := ""
    dir ? while comp := tab(upto('/')) do {
        tab(many('/'))
        d ||= "/" || comp
        if \stat(prefix || d) then {
            # The directory doesn’t exist; create it. d is the
            # directory being copied over; get its uid and mode.
            s := stat(d)
            mkdir(prefix || d, s.mode[2:11])
            chown(prefix || d, s.uid, s.gid)
        }
    }
end

The database file is very simple: for every level, the date of the last backup at that
level is stored. Dates are stored in the system native format so that comparisons with
file modification dates can be easily performed. If no earlier backup is found, procedure
get_time() returns the earliest possible time (the epoch); all files will have newer modified
times, and therefore be backed up.

All the dates found are stored in a global table so that they will be accessible when
backup writes out the database later.

procedure get_time(n)
    # Get the date of earlier backup
    levels := table()
    f := open(dbase)

    while line := read(f) do
        line ? {
            lev := integer(tab(upto(' ')))
            move(1)
            date := tab(0)
            levels[lev] := date
        }
    close(f)
    every i := integer(&digits) do
        if i < n then prev := \levels[i]
        /prev := 0 # default: the epoch
Finally, the program saves the database of dates. It fetches the current time to save with the current level, and deletes all higher-numbered backups from the database.

```unicon
procedure save_time(n)
levels[n] := &now

f := open(dbase, "w") | stop("Couldn't open table ", dbase)
every i := integer(!&digits) do
  if i <= n then write(f, i, " ", \levels[i])
else break
close(f)
end
```

## 15.6 Filtering Email

Unicon’s messaging facilities can be used for filtering email messages obtained from a POP server. A filtering rule has two components: a pattern to look for, and an action to perform if the string is found. The action to be perform can take a few forms: the message can be saved in a folder; it can be deleted; or it can be forwarded to some other address. We can use a record to represent a rule:

```unicon
record rule(pattern, action, args)
```

Here are some example rules represented as records:

```unicon
rule("From: .*spammer\com", "delete")
rule("From: .*unicon-list@lists\.sourceforge\.net", "save", "folders/unicon")
rule("Subject: .*xyzzy", "forward" "shamim@home-domain\.org")
```

We read all the rules from a file and use them to filter all messages. To connect to the POP server, we need some parameters: the server to connect to, the username and the password. These values are stored in a parameter file that we read. The parameters from the file can be stored in a global table called `params`. (We will deal with reading the file later.) First, we connect to the POP server and for every message, invoke the filter:

```unicon
params := get_parameters()
server := \params["server"] | stop("No server specified in .popfilter file")
user := \params["user"] | stop("No user specified in .popfilter file")
password := \params["password"] |
  stop("No password specified in .popfilter file")
```
15.6. FILTERING EMAIL

params["inbox"] | stop("No inbox specified in .popfilter file")

url := "pop://" || user || ":" || password || ":" || server
s := open(url, "m") | stop("Couldn't connect to ", image(url))
while filter(pop(s), rules) do
  # Make sure there are no errors
  if s["Status-Code"] >= 300 then {
    close(s)
    stop(&& progname, ": POP error: ", s["Status-Code"],
    " ", s["Reason-Phrase"] | "")
  }

After reading each message, we make sure that there wasn’t an error in talking to the
POP server – if there was one, we print the error and exit.

The filtering procedure tries to match the message against every rule; if one matches,
it hands the message off to the action procedure. If no rule matches, the message is saved
to the default folder (the inbox).

procedure filter(message, rules)
  local filter_rule
  every filter_rule := !rules do
    if message? ReFind(filter_rule.pattern) then
      perform(filter_rule.action, filter_rule.args, message) |
      write(&errout, "Error in ", image(filter_rule.action),
      " ", image(filter_rule.args))
    # No action matched so we save it in the inbox
    perform("save", params["inbox"], message)
  end

Since the patterns are regular expressions, we link in the IPL regexp package for the
procedure ReFind() that will do the pattern matching.

procedure perform(action, args, message)
  local procname
  procname := \proc("filterproc_" || action, 2) |
  stop(&& progname, ": action ", image(action), " unknown.")
  procname(message, args) | fail
  return ""
end

We use Unicon’s string invocation (and we’d better remember to put invocable all at
the top of the program) to call the procedure associated with each defined action. The
function proc() tries to find the procedure with the right name; if it fails, it is an unknown
action. The names of these procedures all have the form “filterproc_” appened with the
name of the action. We make sure that the procedures succeed if there was no error in performing the action.

```plaintext
procedure filterproc_delete(message, args)
    return "" # We don’t have to do anything
end

procedure filterproc_save(message, filename)
    local f
    f := open(filename, "a") | fail
    write(f, message)
    close(f)
    return ""
end

procedure filterproc_forward(message, address)
    local subject, s

    # Find the subject of the message and use that
    message? (tab(find("Subject: ")) & subject := tab(upto(\n'')))>
    subject := "Subject: " || subject

    s := open("mailto://" || address, "m", subject) | fail
    write(s, message)
    close(s)
    return ""
end
```

It would be easy to implement an action that sends messages to external programs — in procedure `perform()`, if the first character of the action is the pipe symbol “|” we can open a pipe and write the message to it instead of calling the right action procedure. (This would be a good “exercise left to the reader!”)

We still need to read the files and set up the `params` table and the list of rules. It is a good idea to allow blank lines and comments in files we read, since humans will be editing these files and we like comments to remind ourselves what things do. We need a procedure that reads a file and strips out comments and blank lines:

```plaintext
procedure getline(f)
    local line, s
    while line := read(f) do line? {
        s := tab(upto('#') | 0)
        if *trim(s) = 0 then next
        return s
    }
end
```
We read the POP parameters from a file named `.popfilter` in the user’s home directory and return a table of name-value pairs. The global variable `WS` is a cset that contains the space and tab characters; it is initialized in `main()`, when the program first starts executing.

```haskell
# Read the `//.popfilter` file and return a table of name -> value pairs
procedure get_parameters()
    local f, P := table(), line
    f := open(getenv("HOME") || "/.popfilter") | return P
    while line := getline(f) do line ? {
        pname := tab(upto(WS))
        tab(many(WS))
        P[pname] := tab(0)
    }
    close(f)
    return P
end
```

The filtering rules are read from the file named on the command line. The pattern is separated from the action by an exclamation point, and (as usual) comments, blank lines and whitespace (around the “!”) are allowed. Whitespace inside the regular expression is significant and should not be discarded so we use `trim()` to make sure we only remove trailing whitespace. Searches are anchored to the begin of line so we can easily look for individual headers; instead of separating a message into lines and using the “beginning of line” regular expression operator, we simply we add a newline character to the front of the pattern.

```haskell
# Read the file and return a list of rules. All the work is
# actually done in getline() and parse()
procedure read_rulefile(filename)
    local f, rules := [ ]
    f := open(filename) | fail
    while push(rules, parse(getline(f)))
    close(f)
    return rules
end
```

```haskell
# Parse a line into a pattern, an action, and optionally
# arguments to the action.
procedure parse(s)
    local regexp, action

    s ? {
        tab(many(WS))
    }
```
Here is an example action file that includes the example rules from above:

```
# Flush evil spammers!
From: .*spammer\.com !delete
Subject: .*MAKE MONEY FAST !delete # Got enough, thanks!

# Cool people get special attention
From: .*unicon-list@lists\.sourceforge\.net !save folders/unicon
From: .*parlett|jeffery|pereda !save folders/urgent

# The secret password!! Forward it to the pager.
Subject: .*xyzzy !forward pager@shamims-domain.org
```

And here’s an example .popfilter file:

```
# Settings for popfilter.icn, a POP client and email filter
# July 2002

# Where incoming mail is stored locally
inbox /var/mail/shamim

# The POP server
server jantar

# Remember, this is the username and password on jantar
user spm
password Xyz!1234
```

We have introduced the main procedure in pieces; here it is all in one place:

```
link regexp
record rule(pattern, action, args)
global WS

# Email and POP parameters
global params
```
# We're using string invocation
invocable all
procedure main()
    local server, user, password
    WS := ' \t' # whitespace
    params := get_parameters()
    server := params["server"] | stop("No server specified in .popfilter file")
    user := params["user"] | stop("No user specified in .popfilter file")
    password := params["password"] | stop("No password specified in .popfilter file")
    params["inbox"] | stop("No inbox specified in .popfilter file")
    if *args = 0 then stop("Usage: ", &progname, " filter-rule-file")
    url := "pop://" || user || ":" || password || "@" || server
    filter_rules := read_rulefile(args[1]) |
        stop("Couldn't read filter rules from ", image(args[1]))
    filter(testmsg(), filter_rules)
    s := open(url, "m") | stop("Couldn't connect to ", image(url))
    while filter(pop(s), filter_rules) do
        if s["Status-Code"] >= 300 then {
            close(s)
            stop(&progname, ": POP error: ", s["Status-Code"], ": ", s["Reason-Phrase"] | "")
        }
        close(s)
    end

The complete program popfilter.icn can be obtained from the book’s website. There are still a few things this program would need to be truly useful. For example, it should allow non-anchored searches, and the user should be able to specify whether case is significant while looking for a pattern – or perhaps a part of a pattern may be case sensitive by itself. Also, what if we want a “!” in the regular expression? We hope that you will be inspired to make these additions yourself.

15.7 Summary

Writing utilities and system administration tools is easy in Unicon. These programs rely on the system facilities described in Chapter 5. While we do not advocate using Unicon in every case, it is an advantage that your applications language is also an effective scripting language. Ordinary programs can easily take on scripting tasks, and programs that might
otherwise be written in a scripting language have Unicon’s cleaner design and more robust set of control and data structures available.
The Internet is central to modern computing. Because it is ubiquitous, programmers should be able to take it for granted. Writing applications that use the Internet should be just as easy as writing programs for a standalone desktop computer. In many respects this ideal can be achieved in a modern programming language. The core facilities for Internet programming were introduced with simple examples as part of the system interface in Chapter 5. This chapter expands on this important area of software development. This chapter presents examples that show you how to

- Write Internet servers and clients
- Build programs that maintain a common view of multiple users’ actions

16.1 The Client-Server Model

The Internet allows applications to run on multiple connected computers using any topology, but the standard practice is to implement a client/server topology in which a user’s machine plays the role of a client, requesting information or services from remote machines, each of which plays the role of a server. The relationship between clients and servers is many-to-many, since one client can connect to many servers and one server typically handles requests from many clients.

Writing a client can be easy. For simple read-only access to a remote file, it is just as easy as opening a file on the hard disk. Most clients are more involved, sending out requests and receiving replies in some agreed-upon format called a protocol. A protocol may be human readable text or it may be binary, and can consist of any number of messages back and forth between the client and server to transmit the required information. The most common Internet protocols are built-in parts of Unicon’s messaging facilities, but some applications define their own protocol.

Writing a server is more difficult. A server sits around in an infinite loop, waiting for clients and servicing their requests. When only one client is invoking a server, its job is
simple enough, but when many simultaneous clients wish to connect, the server program
must either be very efficient or else the clients will be kept waiting for unacceptably long
periods.

Although the following example programs emphasize how easy it is to write Internet
clients and servers in Unicon, writing "industrial strength" applications requires additional
security considerations which are mostly beyond the scope of this book. For example, user
authentication and encryption are essential in most systems, and many modern servers are
carefully tuned to maximize the number of simultaneous users they support, and minimize
their vulnerability to denial-of-service attacks.

16.2 An Internet Scorecard Server

Many games with numeric scoring systems feature a list of high scores. This feature is
interesting on an individual machine, but it is ten times as interesting on a machine con-
ected to the Internet! The following simple server program allows games to report their
high scores from around the world. This allows players to compete globally. The scorecard
server is called scored. By convention, servers are often given names ending in "d" to
indicate that they are daemon programs that run in the background.

The scorecard client procedure

Before examining the server code, take a look at the client procedure that a game calls to
communicate with the scored server. To use this client procedure in your programs, add
the following declaration to your program.

    link highscor

The procedure highscore() opens a network connection, writes four lines consisting of
the protocol name "HSP", the name of the game, the user’s identification (which could
be a nickname, a number, an e-mail address, or anything else), and that game’s numeric
score. Procedure highscore() then reads the complete list of high scores from the server,
and returns the list. Most games write the list of high scores to a window for the user to
ponder.

    procedure highscore(game, userid, score, server)
      if not find(“:”, server) then server ||= “:4578”
      f := open(server, “n”) | fail

      # Send in this game’s score
      write(f, “HSP\n”, game, “\n\n”, userid, “\n\n”, score) |
      stop(“Couldn’t write: “, &errortext)
# Get the high score list
L := ["High Scores"]
while line := read(f) do put(L, line)
close(f)
return L
end

The Scorecard server program

The scorecard server program, scored.icn illustrates issues inherent in all Internet servers. It must sit at a port, accepting connection requests endlessly. For each connection, a call to \texttt{score_result()} handles the request. The \texttt{main()} procedure given below allows the user to specify a port, or uses a default port if none is supplied. If another server is using a given port, it won’t be available to this server, and the client and server have to agree on which port the server is using.

```
procedure main(av)
    port := 4578 # a random user-level port
    if av[i := 1 to *av] == "-port" then port := integer(av[i+1])

    write("Internet Scorecard version 1.0")
    while net := open(":\| port, "na") do {
        score_result(net)
        close(net)
    }
    (&errno = 0) | stop("scored net accept failed: ", &errortext)
end
```

The procedure \texttt{score_result()} does all the real work of the server, and its implementation is of architectural significance. Any delay in handling a request implies the server will be unable to handle other simultaneous client requests. For this reason, many servers immediately spawn a separate process to handle each request. You could do that with \texttt{system()}, as illustrated in Chapter 5, or launch a thread for it, but for \texttt{scored} this is overkill. The server handles each request almost instantaneously.

Some small concessions to security are in order, even in a trivial example such as this. If a bogus Internet client connects by accident, it will fail to identify our protocol and be rejected. More subtly, if a rogue client opens a connection and writes nothing, we do not want to block waiting for input or the client will deny service to others. A call to \texttt{select()} is used to guarantee the server receives data within the first 1000 milliseconds (1 second). A last security concern is to ensure that the "game" filename supplied is valid; it must be an existing file in the current directory, not something like /etc/passwd for example.

The \texttt{score_result()} procedure maintains a static table of all scores of all games that it knows about. The keys of the table are the names of different games, and the values in
the table are lists of alternating user names and scores. The procedure starts by reading
the game, user, and score from the network connection, and loading the game’s score list
from a local file, if it isn’t in the table already. Both the score lists maintained in memory,
and the high scores files on the server, are sequences of pairs of text lines containing a
userid followed by a numeric score. The high score files have to be created and initialized
manually with some N available (userid,score) pairs of lines, prior to their use by the server.

```
procedure score_result(net)
  local s := ""
  static t, gamenamechars
  initial {
    t := table()
    gamenamechars := &letters++&digits++'-_'
  }

  select(net, 1000) | { write(net, "timeout"); fail }
  (s ||= ready(net)) ? {
    = "HSP\n" | { write(net, "wrong protocol"); fail }
    game := tab(many(gamenamechars)) | { write(net,"no game?"); fail }
    = "\n" 
    owner := tab(many(gamenamechars)) | { write(net,"no owner?"); fail }
    = "\n" 
    score := tab(many(&digits)) | { write("no score?"); fail }
  }

  if t[game] === &null then {
    if not (f := open(game)) then {
      write(net, "No high scores here for ", game)
      fail
    }
    t[game] := L := []
    while put(L, read(f))
    close(f)
  } else
    L := t[game]

  if score > L[-1] then {
```

The central question is whether the new score makes an entry into the high scores list
or not. The new score is checked against the last entry in the high score list, and if it is
larger, it replaces that entry. It is then "bubbled" up to the correct place in the high score
list by repeatedly comparing it with the next higher score, and swapping entries if it is
higher. If the new score made the high score list, the list is written to its file on disk.

```
if score > L[-1] then {
```
16.3. A SIMPLE “TALK” PROGRAM

L[-2] := owner
L[-1] := score
i := -1
while L[i] > L[i-2] do {
    L[i] := L[i-2]
    L[i-1] := L[i-3]
    i := 2
}
f := open(game,"w")
every write(f, !L)
close(f)
}

Note
List L and t[game] refer to the same list, so the change to L here is seen by the next client that looks at t[game].
Lastly, whether the new score made the high score list or not, the high score list is written out on the network connection so that the game can display it.

every write(net, !L)
end

Is this high score application useful and fun? Yes! Is it secure and reliable? No! It records any scores it is given for any game that has a high score file on the server. It is utterly easy to supply false scores. This is an honor system.

16.3 A Simple “Talk” Program

E-mail is the king of all Internet applications. After that, some of the most popular Internet applications are real-time dialogues between friends and strangers. Many on-line services rose to popularity because of their "chat rooms," and Internet Relay Chat (IRC) is a ubiquitous form of free real-time communication. These applications are evolving in multiple directions, such as streaming multimedia, and textual and graphical forms of interactive virtual reality. While it is possible to create arbitrarily complex forms of real-time communication over the Internet, for many purposes, a simple connection between two users’ displays, with each able to see what the other types, is all that is needed.

The next example program, called italk, is styled after the classic BSD UNIX talk program. The stuff you type appears on the lower half of the window, and the remote party’s input is in the upper half. Unlike a chat program, the characters appear as they are typed, instead of a line at a time. In many cases this allows the communication to occur more smoothly with fewer keystrokes.
The program starts out innocently enough, by linking in library functions for graphics, defining symbolic constants for font and screen size. Among global variables, vs stands for vertical space, cwidth is column width, wheight and wwidth are the window’s dimensions, and net is the Internet connection to the remote machine.

```
link graphics
$define ps 10  # The size of the font to use
$define lines 48  # No. of text lines in the window
$define margin 3  # Space to leave around the margins
$define START_PORT 1234 $define STOP_PORT 1299
global vs, cwidth, wheight, wwidth, net
```

The main() procedure starts by calling win_init() and net_init() to open up a local window and then establish a connection over the network, respectively. The first command line argument is the user and/or machine to connect to.

```
procedure main(args)
    win_init()
    net_init(args[1] | "127.0.0.1")
end
```

Before describing the window interaction or subsequent handling of network and window system events, consider how italk establishes communication in procedure net_init(). Unlike many Internet applications, italk does not use a conventional client/server architecture in which a server daemon is always running in the background. To connect to someone on another machine, you name him or her in the format user@host on the command line. The code attempts to connect as a client to someone waiting on the other machine, and if it fails, it acts as a server on the local machine and waits for the remote party to connect to it.

```
procedure net_init(host)
    host ?:= {
        if user := tab(find("@")) then move(1)
        tab(0)
    }
    net := (open_client|open_server|give_up)(host, user)
end
```

The attempt to establish a network connection begins by attempting to open a connection to an italk that is already running on the remote machine. The italk program works with an arbitrary user-level set of ports (defined above as the range 1234-1299). An italk client wades through these ports on the remote machine, trying to establish a connection with the desired party. For each port at which open() succeeds, the client writes its user name, reads the user name for the process on the remote machine, and returns the connection if the desired party is found.
procedure open_client(host, user)
    port := START_PORT
    if \user then {
        while net := open(host || ":" || port, "n") do {
            write(net, getenv("USER") | "anonymous")
            if user == read(net) then return net
            close(net)
            port += 1
        }
    }
    else {
        net := open(host || ":" || port, "n")
        write(net, getenv("USER") | "anonymous")
        read(net)  # discard
        return net
    }
end

The procedure open_server() similarly cycles through the ports, looking for an available one on which to wait. When it receives a connection, it checks the client user and returns the connection if the desired party is calling.

procedure open_server(host, user)
    repeat {
        port := START_PORT
        until net := open(":" || port, "na") do {
            port += 1
            if port > STOP_PORT then fail
        }
        if not (them := read(net)) then {
            close(net)
            next
        }
        if /user | (them == user) then {
            write(net, getenv("USER") | "anonymous")
            WAttrib("label=talk: accepted call from ", them)
            return net
        }
        WAttrib("label=talk: rejected call from ", them)
        write(net, getenv("USER") | "anonymous")
        close(net)
    }
end

This connection protocol works in the common case, but is error prone. For example,
if both users typed commands at identical instants, both would attempt to be clients, fail, and then become servers awaiting the other’s call. Perhaps worse from some users’ point of view would be the fact that there is no real authentication of the identity of the users. The italk program uses whatever is in the USER environment variable. The UNIX talk program solves both of these problems by writing a separate talk server daemon that performs the marshalling. The daemons talk across the network and negotiate a connection, check to see if the user is logged in and if so, splash a message on the remote user’s screen inviting her to start up the talk program with the first user’s address.

The next part of italk’s code to consider is the event handling. In reality, each italk program manages and multiplexes asynchronous input from two connections: the window and the network. The built-in function that is used for this purpose is select(). The select() function will wait until some (perhaps partial) input is available on one of its file, window, or network arguments.

The main thing to remember when handling input from multiple sources is that you must not block for I/O. This means: use listener mode for new connections or a timeout parameter with open(), and when handling network connections, never use read(), only use reads() or better yet ready(). For windows you must also avoid read()’s library procedure counterpart, WRead(). The code below checks which connection has input available and calls Event() as events come in on the window, and calls reads() on the network as input becomes available on it. In either case the received input is echoed to the correct location on the screen. Ctrl-D exits the program. To accept a command such as "quit" would have meant collecting characters till you have a complete line, which seems like overkill for such a simple application.

```c
repeat {
  *(L := select(net, &window))>0 | stop("empty select?")
  if L[1] == &window then {
    if &lpress >= integer(e := Event()) >= &rdrag then next
    if string(e) then {
      writes(net, e) | break
      handle_char(2, e) | break
      WSync()
    }
  }
  else {
    s := reads(net) | break
    handle_char(1, s) | break
    WSync()
  }
}
close(net)
end
```
After such a dramatic example of input processing, the rest of the \texttt{italk} program is a bit anticlimactic, but it is presented anyhow for completeness sake. The remaining procedures are all concerned with managing the contents of the user’s window. Procedure \texttt{handle\_char(w, c)}, called from the input processing code above, writes a character to the appropriate part of the window. If \( w = 1 \) the character is written to the upper half of the window. Otherwise, it is written to the lower half. The two halves of the window are scrolled separately, as needed.

\begin{verbatim}
procedure handle\_char(w, c)
  # Current horiz. position for each half of the window
  static xpos
  initial xpos := [margin, margin]
  if \texttt{c} == "\d" then fail  # EOF

  # Find the half of the window to use
  y\_offset := (w - 1) \ast \texttt{wheight}/2

  if \texttt{c} == ("\r"|\n') \texttt{\&} xpos[w] > wwidth then {
    ScrollUp(y\_offset+1, w\texttt{wheight}/2-1)
    xpos[w] := margin
  }
  if \texttt{c} == ("\r"|\n') then return

  # handles backspacing on the current line
  if \texttt{c} == "\b" then {
    if xpos[w] >= margin + cwidth then {
      EraseArea(xpos[w]-cwidth, y\_offset+1+w\texttt{wheight}/2-1-vs, cwidth, vs)
      xpos[w] := cwidth
      return
    }
  }

  DrawString(xpos[w], w\texttt{wheight}/2 + y\_offset - margin, c)
  xpos[w] +:= cwidth
  return
end

Scrolling either half of the window is done a line at a time. The graphics procedure \texttt{CopyArea()} is used to move the existing contents up one line, after which \texttt{EraseArea()} clears the line at the bottom.

\begin{verbatim}
procedure ScrollUp(vpos, h)
  CopyArea(0, vpos + vs, wwidth, h-vs, 0, vpos)
  EraseArea(0, vpos + h - vs, wwidth, vs)
end
\end{verbatim}
The window is initialized with a call to the library procedure, \texttt{WOpen()}, which takes attribute parameters for the window’s size and font. These values, supplied as defined symbols at the top of the program, are also used to initialize several global variables such as \texttt{vs}, which gives the vertical space in pixels between lines.

```pascal
procedure win_init()
    WOpen("font=typewriter," || ps, "lines=" || lines, "columns=80")
    wwidth := WAttrib("width")
    wheight := WAttrib("height")
    vs := WAttrib("fheight")
    cwidth := WAttrib("fwidth")
    DrawLine(0, wheight/2, wwidth, wheight/2)
    Event()
end
```

Lastly, the procedure \texttt{give_up()} writes a message and exits the program, if no network connection is established. If \texttt{user} is null and the non-null test (the backslash operator) fails, the concatenation is not performed and alternation causes the empty string to be passed as the second argument to \texttt{stop()}.

```pascal
procedure give_up(host, user)
    stop("no connection to ", (user || "@") | ",", host)
end
```

What enhancements would make \texttt{italk} more interesting? An obvious extension would be to use a standard network protocol, such as that of UNIX \texttt{talk}, so that \texttt{italk} could communicate with other users that don’t have \texttt{italk}. UNIX \texttt{talk} also offers a more robust connection and authentication model (although you are dependent on the administrator of a remote machine to guarantee that its \texttt{talkd} server is well behaved). Another feature of UNIX \texttt{talk} is support for multiple simultaneously connected users.

One neat extension you might implement is support for graphics, turning \texttt{italk} into a distributed whiteboard application for computer-supported cooperative work. To support graphics you would need to extend the window input processing to include a simple drawing program, and then you would need to extend the network protocol to include graphics commands, not just keystrokes. One way to do this would be to represent each user action (a keystroke or a graphics command) by a single line of text that is transmitted over the network. Such lines might look like:

```
key H
key i
key !
circle 100,100,25
```
and so forth. At the other end, the program deciphering these commands translates them into appropriate output to the window, which would be pretty easy, at least for simple graphics. The nice part about this solution is that this particular collaborative whiteboard application would work fine across differing platforms (Linux, Microsoft Windows, and so on) and require only a couple hundred lines of code!

16.4 Summary

Writing Internet programs can be easy and fun, although it is easy to underestimate the security needed. There are several different ways to write Internet programs in Unicon. The database interface presented in Chapter 6 allows you to develop client/server applications without any explicit network programming when the server is a database. A SQL server is overkill for many applications such as the high score server, and it is not appropriate for other non-database network applications such as the italk program.

For these kinds of programs, it is better to "roll your own" network application protocol. Once a connection is established (perhaps using a client/server paradigm), the actual communication between programs is just as easy as file input and output. If you do roll your own network application, keep the protocol simple; it is easy enough to write yourself into deadlocks, race conditions, and all the other classic situations that make parallel and distributed programming perilous.
Chapter 17

Genetic Algorithms

The previous three chapters showed you how to write Unicon programs with many kinds of Internet and system capabilities that are expected of most modern applications. Unicon is great for generic computing tasks, but it really excels when its advanced features are applied in application areas where the development of the algorithms is complex. This chapter describes how to use Unicon to build an entire, somewhat complex application with reusable parts. The field of genetic algorithms (GAs) is an exciting application domain with lots of opportunities for exploratory programming. When you’re finished with this chapter, you will

- Understand the basics of genetic algorithms.
- See how to build genetic algorithm engine in Unicon.
- Use that GA engine to build programs for your own projects.

17.1 What are Genetic Algorithms?

The broad field of evolutionary computation has been an area of active research since the 1950s. Initially, it was led by computer scientists that believed evolution could be used as an optimization tool for engineering problems. Genetic Programming (GP) focuses on evolving computer programs to perform various tasks. On the other hand, GAs focus on the simpler task of evolving data that is used to solve a problem. The GAs that we’ll study have a binary representation. Increasingly there has been a shift towards non-binary representations such as floating-point numbers in GA-related projects. That field has typically been given the more general name of evolutionary algorithms. GAs have one of the most well-defined mathematical foundations in all of evolutionary computation, and are a good place to start exploring.

John Holland invented the first GAs in the 1960s. His goal was to study adaptation as it occurs in nature and then to create computer systems to model the adaptive process.
Holland combined four elements that are common to all GAs:

- A population of individuals
- Selection based on fitness
- Mating of individuals
- Random mutation

Consider the very simple problem of finding the largest number encoding in binary with six digits. Assume the GA knows nothing about binary encoding.

While making use of a population might seem to be a necessary element for any evolutionary computation, it is not. Instead, you could focus all your efforts on improving one individual. In this case, fitness is exactly the numerical value of an individual. For example, you could examine the fitness of this one individual with an exhaustive search:

```unicon
best := 0
every i := 0 to 2^6 do
    best <:= i
```

Suppose you only make use of the elements of a population and selection based on fitness. You could have a population of six individuals, randomly initialized, and you could attempt to improve the overall fitness of the six by replacing the lowest fitness individual with a random one. The code below shows how to implement this idea:

```unicon
maxi := 2^6
every i := 1 to 100 do {
    worst := 1
    every i := 1 to 5 do
        if population[worst] <= population[i] then
            worst := i
    population[worst] := ?maxi
}
```

Before modeling mating and mutation, you’ll have to create a more detailed representation of the internals of an individual.

### 17.2 Operations: Fitness, Crossover, and Mutation

An individual is represented by a string from a binary alphabet. Incidentally, natural evolution of DNA is based on a quaternary alphabet, but the size of the alphabet is unimportant for a computer model. In Unicon, you could represent these individuals with lists of integers. However, strings of "1" and "0" characters provide a representation that is easier to use. So, here is a more explicit representation of a population:
17.2. OPERATIONS: FITNESS, Crossover, AND Mutation

population := ["010111", "000101", "111101", "111011", "111110", "010110"]

**Fitness**

The fitness can be computed by converting the string representation into an integer as follows: `integer(2r || population[1])`. The `2r` means that this is a literal representation of an integer in base two form. There are many different possible selection schemes used in GAs. This chapter uses one that has proven to be very robust in a large number of different GA applications, called tournament selection. The general idea is to group the individuals and have them compete head-to-head. The winners of the tournaments are selected to live in the next generation; their bits are copied into an element in the new population. Tournaments of size two work well. All you must do is randomly pair up the individuals, and move the one with the higher fitness to the next generation. Because you generally will want the population size to remain constant, you’ll have to do this pairing twice. Here is a tournament selection on the above population:

\[
\begin{align*}
\text{population}[1] & := "010111" \ # 23 \ \text{winner} \\
\text{population}[6] & := "010110" \ # 22 \\
\text{population}[3] & := "111101" \ # 61 \ \text{winner} \\
\text{population}[4] & := "111011" \ # 59 \\
\text{population}[5] & := "111110" \ # 62 \ \text{winner} \\
\text{population}[2] & := "000101" \ # 5
\end{align*}
\]

The second round of selections is listed here:

\[
\begin{align*}
\text{population}[5] & := "111110" \ # 62 \ \text{winner} \\
\text{population}[6] & := "010110" \ # 22 \\
\text{population}[2] & := "000101" \ # 5 \\
\text{population}[3] & := "111101" \ # 61 \ \text{winner} \\
\text{population}[4] & := "111011" \ # 59 \ \text{winner} \\
\text{population}[1] & := "010111" \ # 23
\end{align*}
\]

The end result of tournament selection is listed here:

\[
\begin{align*}
\text{next_gen}[1] & := \text{population}[1] := "010111" \ # 23 \\
\text{next_gen}[2] & := \text{population}[3] := "111101" \ # 61 \\
\text{next_gen}[3] & := \text{population}[5] := "111110" \ # 62 \\
\text{next_gen}[4] & := \text{population}[5] := "111110" \ # 62 \\
\text{next_gen}[5] & := \text{population}[3] := "111101" \ # 61 \\
\text{next_gen}[6] & := \text{population}[4] := "111011" \ # 59
\end{align*}
\]

Notice how there are two copies of \text{population}[3] and \text{population}[5] in \text{next_gen}. On the other hand, there are no copies of \text{population}[2].
Crossover

Mating, more technically known as crossover, involves the sharing of information between members of the population. Again there are many different types of mating schemes, but this chapter describes one called two-point crossover (Figure 17-1) that has proven to be very robust for a wide range of GA applications. Once again, randomly pair up the individuals, but this time instead of competing, individuals will mate. First you must transform the linear strings into circular rings. For each pair, randomly select two points in the ring and cut the rings at the two selected points. Then swap the ring segments to form two new rings.

Figure 17-1: Two-point crossover.

The code for two-point crossover is presented below. The \( ?(l_{\text{chrom}}+1) \) expression picks a random number between one and the length of the chromosome. Variables \( a \) and \( b \) are initialized to two different values in this range; \( a \) is made the smaller of the two indices. Two children in the new generation are formed by splicing portions of \( \text{parent1} \) and \( \text{parent2} \) within the range from \( a \) to \( b \).

```plaintext
a := ?(l_{\text{chrom}}+1)
while ((b := ?(l_{\text{chrom}}+1)) = a)
    if a > b then a :=: b
    n_{\text{cross}} :=+ 1
```
17.3. THE GA PROCESS

Mutation

The last GA operation is mutation. Mutation works at the independent level of single binary digits. To implement mutation, take a look at each bit of each individual of the population. With a fixed probability, flip the value of the bit. This is the basic mechanism for injecting completely new information into the population. Almost all of that information will be useless, but as the GA evolves it will weed out the useless information and keep the useful information.

17.3 The GA Process

Now that you have a handle on the basic operations, it is time to describe the basic GA algorithm for applying these operations.

1. Generate a random population of n individuals each with l-bits.
2. Calculate the fitness of each individual.
3. Perform tournament selection on the population to produce a new generation.
4. With probability $p_c$, mate pairs of individuals using two-point crossover.
5. With probability $p_m$, mutate the bits in the population.
6. Replace the old population with the new generation.
7. Go to step 2, until the population meets some desired condition.

Even with this mechanical algorithm, applying a GA to any specific problem remains an art. For example, step 7 leaves the desired stopping condition up to the implementer. Typically the stopping condition might be something like: until the average fitness has not risen significantly in the last five generations. This is one of many implementation choices you have to make, from variations on crossover to adjusting the mutation and mating rates. Here are some time-tested rules of thumb:

1. Encode the solutions to a problem with as few bits as possible but not at the expense of making the encoding very complex.
2. Let the size of the population be at least twenty but not so large that your computer program is intolerably slow.
3. Mating rates between 30 percent and 90 percent work for a large range of problems.
4. Mutation rates should be near 1/(the number of bits), so that each individual undergoes about one mutation on average per generation.
5. Once the average fitness of the population does not change significantly after ten generations, the population has converged on the solution. At this point, stop the GA and study the population of solutions.
17.4  **ga_eng**: a Genetic Algorithm Engine

The **ga_eng** engine is a general purpose reusable GA engine that can quickly and easily be adapted to solve a variety of problems. This section presents its key elements. The full source code for **ga_eng** is on the book’s web site.

From the preceding sections, you can tell that a GA maintains a large amount of information as it transitions from one generation to the next. What is the state information that the engine needs to track? Below is a list of the most obvious things to record:

- **n** — the size of the population
- **l** — the length of the individual’s bit representation
- **p_c** — the probability of crossover mating
- **p_m** — the probability of mutation
- **population** — a list of a current population’s individuals

Given the above state information, what can a GA engine do?

- **init()** — initialize the state information and population
- **evolve()** — move a population from one generation to the next
- **tselect()** — perform tournament of selection on the population
- **stats()** — collect statistics about the current population to monitor progress

The fitness function

A key application-specific interface issue is the fitness function. The user of the GA engine sets the fitness values of each member of the population. If that value is not set, the default value will be the average of the fitness of each of the parents. Each parent’s contribution to the fitness is weighted by how many bits it contributed to the offspring. This has the nice property of not requiring that each individual’s fitness be computed at every generation.

To use the GA engine, the programmer supplies an application-specific fitness function **f(x)** that is applied to each individual of the population every generation. Given a binary string **s**, **f(s)** would return a numeric fitness value.
Methods and attributes of class \texttt{ga\_eng}

Class \texttt{ga\_eng} implements the engine. It provides two public methods, \texttt{evolve()} and \texttt{set\_params()}, as well as eight private methods:

\begin{verbatim}
method evolve()
method set\_params(fitness\_func, popsize, lchrom, pcross, pmutation, log\_file)
method tselect()
method crossover2(parent1, parent2)
method generation()
method stats(Pop)
method report(Pop)
method random\_chrom()
method initpop()
method init()
\end{verbatim}

The \texttt{set\_params()} method sets the fitness function, the population size, the length of the chromosomes, the probability of crossover, the probability of mutation, and the log file for generating a trace of a run. The constructor \texttt{ga\_eng()} takes the same input parameters as \texttt{set\_params()} but it also re-initializes the engine by creating a new population from scratch. The \texttt{evolve()} method moves the population from one generation to the next.

The \texttt{tselect()} method operates on the whole population by performing tournament selection. The \texttt{crossover2()} method takes two individuals and returns two new individuals in a list after doing two-point crossover. The \texttt{stats()} method collects statistics on the given population, and \texttt{report()} writes the results out to the log file if it is not set to \texttt{&null}. The \texttt{init()} and \texttt{initpop()} methods initialize the GA.

You might be asking what is a population of individuals? The key pieces of information about an individual are its chromosomes, which are represented as a string of zeros and ones. For implementation reasons, we bundle the following information for each individual using the following Unicon record:

\begin{verbatim}
record individual(chrom, fitness, parent1, parent2, xsite)
\end{verbatim}

This stores the fitness values, the index of two parents, and the crossover sites from when the parents were mated.

The class \texttt{ga\_eng} makes use of the following instance variables:

\begin{itemize}
\item \texttt{oldpop, newpop} — two populations; selection goes from \texttt{oldpop} into \texttt{newpop}
\item \texttt{popsize, lchrom} — the population size and the length of the chromosomes
\item \texttt{gen, maxgen} — current and max generation number
\item \texttt{pcross, pmutation} — the probability of crossover and mutation
\end{itemize}
• **sumfitness** — the sum of the fitness of the entire population

• **nmutation** — number of mutation in the current generation

• **ncross** — number of crossovers (or matings) in the current generation

• **avg, max, min** — average, maximum, and minimum fitness in the population

• **best** — the location of the individual with the highest fitness

• **log_file** — a text file where statistics are written during a GA run

• **fitness_func** — the user-supplied fitness function. It reads a string of zeros and ones and returns a number representing the fitness.

### A Closer Look at the `evolve()` method

Space limitations preclude discussing every method of `ga_eng`, but `evolve()` is a method that is called by user code, and defines the basic architecture of the engine. The `evolve()` method initializes the population the first time the method is invoked by each instance of `ga_eng`. After initialization and in subsequent calls, `evolve()` does three things: it collects statistics, writes the results to a log file, and then evolves the population for one generation. The method `generation()` then becomes the focus of activity. The code for `evolve()` is:

```plaintext
method evolve()
    if /initialized := 1 then {
        gen := 0
        init()
        statistics(oldpop)
        if \log_file then report(oldpop)
    }
    gen +:= 1
    generation()
    statistics(newpop)
    if \log_file then report(newpop)
    oldpop := newpop
end
```

Generation consists of three high-level operations. Tournament selection is performed via a call to `tselect()`, which makes copies of selected individuals from `oldpop` to `newpop`. After this, all operations take place on individuals in `newpop`. The next two operations are performed in a loop, on pairs of individuals. The `crossover2()` method does the mating; it encapsulates the relatively low-level operation of mutation. The last high-level operation that a GA does is call the user supplied `fitness_func` to assign a fitness value to each individual in the new population. The GA keeps track of only two generations for the
evolve() method to continue as long as needed. Once each individual has been evaluated, that generation is complete. The oldpop variable is assigned the value of the newpop, and the process is ready to start again. Listing 17-1 shows the code for the method generation():

Listing 17-1
A method for producing a new generation.

method generation()
  local j := 1, mate1, mate2, jcross, kids, x, fitness1, fitness2, selected
  newpop := list(popsize)
  nmutation := ncross := 0
  selected := tselect()
  repeat {
    mate1 := selected[j]
    mate2 := selected[j+1]
    kids := crossover2(oldpop[mate1].chrom, oldpop[mate2].chrom)
    fitness1 := fitness_func(oldpop[mate1].chrom)
    fitness2 := fitness_func(oldpop[mate2].chrom)
    newpop[j] := individual(kids[1], fitness1, mate1, mate2, kids[3])
    newpop[j+1] := individual(kids[2], fitness2, mate1, mate2, kids[3])
    if j > popsize then break
    j +:= 2
  }
end

Using ga_eng

GAs are extremely robust. It is easy to create a buggy GA that works so well that the bugs go undetected. Masking of bugs by robust algorithms is not unique to GA; it occurs in many numerical algorithms. To prevent this, the following code tests the GA on a simple problem where having one bit increases the fitness values by one. This is a ready to compile and run GA application, albeit a very simple one. As you can see, ga_eng() and evolve() are all the interface methods you need to build a complete genetic algorithm. Listing 17-2 shows all the code needed to use the GA engine for a simple test problem.

Listing 17-2
Using ga_eng() for a simple test problem

procedure decode(chrom)
  local i := 0
  every !chrom == "1" do i +:= 1
  return i
end
# a simple test of the engine, fitness is based on number of 1 bits in chrom

```plaintext
procedure main()
    log_file := open("test_ga.log", "w") | stop("cannot open log_file.log")
ga := ga_eng(decode, 100, 20, 0.99, 1.0/real(20), log_file)
every 1 to 100 do ga.evolve()
    write(log_file, "best location => ", ga.best)
    write(log_file, "best fitness => ", ga.newpop[ga.best].fitness)
end
```

Log files

The log file `test_ga.log` contains a lot of statistics. Listing 17-3 is a fragment of the log file that is generated. The complete log file has over twelve thousand lines.

Listing 17-3
A trace of the GA for a simple test problem

Log File for Genetic Algorithm (GA)

<table>
<thead>
<tr>
<th>Population size</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromosome length</td>
<td>20</td>
</tr>
<tr>
<td>Maximum # of generations</td>
<td></td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.99</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Initial population maximum fitness = 5.00e-1
Initial population average fitness = 5.00e-1
Initial population minimum fitness = 5.00e-1
Initial population sum of fitness = 5.00e1

-----------------------------------------------
Population Report

<table>
<thead>
<tr>
<th>Generation 0</th>
</tr>
</thead>
<tbody>
<tr>
<td># parents</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>1) ( 0, 0)</td>
</tr>
<tr>
<td>2) ( 0, 0)</td>
</tr>
<tr>
<td>3) ( 0, 0)</td>
</tr>
<tr>
<td>..............</td>
</tr>
<tr>
<td>100) ( 0, 0)</td>
</tr>
</tbody>
</table>

-----------------------------------------------
Statistics:
min = 5.00000000e-1  avg = 5.00000000e-1  max = 5.00000000e-1
no. of mutations = 0
17.5. Color Breeder: a GA Application

Normally, the human eye is capable of distinguishing millions of different colors. If you have a device capable of producing a large number of different colors such as a PC with a color monitor or color printer, then finding exactly the color that you have in mind can be a tricky task. The color space is quite large. For example, imagine that you want to create a Web page with a blue background. But this is no ordinary blue; you want a blue that is like the blue you saw on the Mediterranean skyline on your last trip to Greece.

One option is to look at a color wheel and click on the part that represents the blue that you have in mind. Unfortunately, the color you choose is typically only a very small part of the color wheel. Also the color is surrounded by many different colors that may be distracting or misleading in your evaluation. When you use the same color for the entire background, you get a different sense of the color.

A second option is to tinker with numeric color codes by hand. Through trial and error, by examining a large number of colors you can select the right one. This can be very time consuming and you may become impatient and only experiment with a small number of
colors, in which case you settle for a quick approximation of the color you intended.

The following color breeder program has properties from both of the color wheel and the tinkering methods. It uses a GA to explore the problem space. The program cb first displays sixteen randomly generated colors. The fitness of an individual color is based entirely on user preference. Using scrollbars, the user ranks the individuals, and then hits a breed button to generate a new population of colors.

Scrollbars sbar\[i\] are indexed with \(i\), where \(i\) runs from 1 to 16. The value of the scrollbar is inverted because scrollbar values correspond to the y-axis by default, which grows from top to bottom; when the tab is lowered the value of scrollbar goes up. The user of cb sets higher tabs to indicate higher fitness.

\[
every\ i:=1\ to\ 16\ do\ \{
\quad f:=VGetState(widgets["\text{"sbar"}||i])
\quad ga.newpop[i].fitness:=1-f
\}\n\]

Gradually the population evolves from a set of random colors to a set of colors that look alike, with ever so slight variations on that Mediterranean sky blue you were thinking of. You can then save a snapshot of the screen in a GIF format if you wish, and can see the numeric codes that represents the color by clicking on a color.

Figure 17-2 is a screenshot of cb. The higher the user slides the scrollbar tab, the more fit the color. There are three color resolutions to select from: 12-, 24-, and 48-bit. The bits are equally divided into three segments, each representing a color intensity. There are two mutually exclusive advanced modes: patterns and text. Patterns are square bit patterns that specify the mix of foreground and background colors in three resolutions: 6x6, 8x8, and 10x10. In text mode, the foreground color is displayed as text atop the background.

Figure 17-2: cb - a genetic algorithm for picking colors.
Breeding textures

The user can turn on bi-level patterns of the form: \texttt{width, #data} as described in Chapter 7. A bi-level pattern is tiled using the foreground and background color to fill an area; a 1 in the pattern denotes the foreground, and a 0 denotes the background. The patterns come in three resolutions: 8x8, 10x10, and 12x12. In low resolutions it is hard to find interesting patterns; in high resolution, most patterns look like a random mixture and are hard to evolve into something more structured. Figure 17-3 (left) shows \texttt{cb} in pattern mode.

![Figure 17-3: cb in pattern mode (left) and text mode (right)](image)

The user can click on the snap shot button to create a GIF image file that is a snapshot of the whole screen. If you left click on a color region, the color and patterns specification will be displayed in a pop window. For the sake of compactness, the colors and patterns are represented with hexadecimal strings as opposed to binary strings.

17.6 Picking Colors for Text Displays

Perhaps the most fun use of \texttt{cb} is to pick colors for your Web pages. Figure 17-3 (right) is a snapshot of \texttt{cb} in text mode. In text mode with 24-bit color, the user can right click on a color region to generate a sample HTML document such as the one shown in Figure 17-4 that demonstrates how to incorporate the selected color combination inside of a Web page. It is also possible to get the hexadecimal representation of a pattern or a color by right clicking on it.
Summary

GAs are a branch of evolutionary computation that allow a general-purpose optimization technique. The three main GA operations are selection, mating, and mutation. Object-oriented techniques enable a generic GA engine that can be adapted to a large variety of problems. By making the engine very general purpose it is possible to create applications with novel properties like using user preferences to set the fitness of a color!
Chapter 18

Object-oriented User Interfaces

Many applications interact with users through a graphical user interface. While Unicon’s graphics facilities are excellent for drawing to the screen, the standard elements of graphical user interfaces are not built-in to the language.

This chapter presents a user interface class library originally developed by the incomparable Robert Parlett, who wrote the first version of this chapter. Object-oriented design is employed to reduce complexity, resulting in an elegant, extensible library that is accessed by importing the package gui. Although this book describes the components provided by the GUI toolkit, you may need the book Graphics Programming in Icon [Griswold98] to develop advanced custom user interfaces with application-specific graphics.

The GUI classes are supported by a tool named ivib that allows interfaces to be constructed by drawing a dialog on the screen interactively. ivib generates a Unicon program that can be filled in to create an application. This chapter shows how to:

- Construct programs that employ a graphical user interface.
- Manipulate the attributes of objects such as buttons and scrollbars.
- Draw a program’s interface using Unicon’s improved visual interface builder.

18.1 A Simple Dialog Example

Object-orientation seems to be a big help in designing graphical user interfaces. The best way to see how the GUI classes work is to try out a simple example program. Listing 18-1 shows the source code in full; the code is explained in detail below.

Listing 18-1
The TestDialog Program

import gui
$include "guih.icn"
class TestDialog : Dialog()
  method component_setup()
    local b, l := Label("label=Click to close","pos=50%,33%","align=c,c")
    add(l)
    b := TextButton("label=Button", "pos=50%,66%", "align=c,c")
    b.connect(self, "dispose", ACTION_EVENT)
    add(b)
    attrib("size=215,150", "bg=light gray", "font=serif", "resize=on")
  end
end

procedure main()
  TestDialog().show_modal()
end

If the program is stored in a file testdialog.icn, the following command will compile it:

    unicon testdialog

The result should be an executable file called testdialog. This example program, along with several others, is in the guidemos directory of the Unicon distribution. Run this program, and the window shown in Figure 18-1 appears; it closes when the button is clicked.

Figure 18-1: TestDialog window

This example program begins by declaring a class, TestDialog. The line

    class TestDialog : Dialog()

indicates that TestDialog is a subclass of the class Dialog that is defined in the toolkit. This subclass relationship is true of all dialog windows.

The remainder of the class’s code is contained in a single method, component_setup(). This method is invoked by the toolkit; it is a convenient place to setup the dialog’s content. Inside the component_setup() method is the code that adds the label to the dialog:
18.2. POSITIONING OBJECTS

l := Label("label=Click to close","pos=50%,33%","align=c,c")
add(l)

This assigns l to a new Label object and sets the label string. The horizontal position is set to 50 percent of the window width and the vertical to 33 percent of the window height. The alignment of the object is centered both vertically and horizontally about the position. Finally, the label is added to the dialog with the line add(l).

The code to add a button is very similar, but a TextButton object is created rather than a Label object, and the vertical position is 66 percent of the window height.

The next line is more interesting:

b.connect(self, "dispose", ACTION_EVENT)

This adds a listener to the button, and tells the toolkit than whenever the button fires an ACTION_EVENT, which it will when it is pressed, the dispose() method in the class self, should be invoked. self of course refers to the TestDialog class, and dispose() is a method inherited from the base class Dialog, which simply closes the window. So all this means is that when the button is pressed, the dialog will close.

The next line sets the attributes of the dialog window, including its initial size. Try changing these values to experiment with other dialog styles.

attrib("size=215,150", "bg=light gray", "font=serif", "resize=on")

After the class comes a standard Icon main() procedure. This simply creates an instance of the dialog and invokes the method show_modal(). This call displays the dialog window and goes into the toolkit’s event handling loop.

18.2 Positioning Objects

The button and label were positioned above using percentages of the window size. An object can also be positioned by giving an absolute position, or by giving a percentage plus or minus an offset. So the following are all valid position specifiers:

"100"
"10%"
"25%+10"
"33%-10"

Positions are often specified in constructors with x and y values separated by commas. By default, positions are relative to the top left corner of the object. The "align" attribute, which takes two alignment specifiers, changes this default. The first alignment specifier is an "l", "c", or "r", for left, center, or right horizontal alignment, respectively; the second
CHAPTER 18. OBJECT-ORIENTED USER INTERFACES

is a "t", "c", or "b", for top, center, or bottom vertical alignment, respectively. Another attribute, "size", has specifiers that take the same format as the position attribute. Most of the toolkit objects default to sensible sizes, and the size attribute can often be omitted. For example, a button’s size defaults fit its string label based on the font in use.

The method attrib(...), in utility class SetFields, implements attribute processing, mostly farming out work to special-purpose methods that can be called directly: set_pos(x,y), set_label(s), set_align(horizontal, vertical), and set_size(w, h). Other than these setter methods, Icon graphics attributes can be interspersed. For example, the attribute "bg=green" will set the object’s background color.

Here are some examples of position, alignment, and size parameters, and a description of their meaning. In the call

attrib("pos=50%,100", "align=c,t", "size=80%,200")

the object is centered horizontally in the window, using 80 percent of the width; vertically its top starts at 100 and its height is 200 pixels. In contrast, the code

attrib("pos=100%,100%, "align=r,b", "size=50%,50%")

specifies that the object fills up the bottom right quarter of the window. The call

attrib("pos=33%+20,0", "size=100,100%")

directs that the object’s left hand side is at one-third of the window size plus 20 pixels; it is 100 pixels wide. It fills the whole window vertically.

18.3 A More Complex Dialog Example

Now it’s time to introduce some more component types. Listing 18-2 shows our next example program in full.

Listing 18-2
SecondTest Program

import gui
$include "guih.icn"
class SecondTest : Dialog(
    # Class variables representing objects in the dialog.
    text_list, table, list, text_field,
    oses, languages, shares     # Some data variables.
)
# Add a line to the end of the text list
# method put_line(s)
  local l := text_list.get_contents()
  put(l, s)
  text_list.set_contents(l)
  text_list.goto_pos("l")
end

# Event handlers - produce a line of interest.
# method handle_check_box_1(ev)
  put_line("Favorite OS is " || oses[1])
end
method handle_check_box_2(ev)
  put_line("Favorite OS is " || oses[2])
end
method handle_check_box_3(ev)
  put_line("Favorite OS is " || oses[3])
end
method handle_text_field(ev)
  put_line("Contents = " || text_field.get_contents())
end
method handle_list(ev)
  put_line("Favorite language is " || languages[list.get_selection()])
end
method handle_text_menu_item_2(ev)
  put_line("You selected the menu item")
end

# The quit menu item
# method handle_quit(ev)
  dispose()
end

method handle_table(ev)
  local i := table.get_selections()[1]
  put_line(shares[i][1] || " is trading at " || shares[i][2])
end
method handle_table_column_1(ev)
    put_line("Clicked on column 1")
end

method handle_table_column_2(ev)
    put_line("Clicked on column 2")
end

#
# This method is invoked for a component which may potentially want
# to handle an event (by firing an event to its listeners for
# example). A dialog is just another custom component, and so it can
# override this method to do any custom processing.
#
method handle_event(ev)
    put_line("Icon event " || ev)
    self.Dialog.handle_event(ev)
end

method component_setup()
    local menu_bar, menu, panel_1, panel_2, panel_3, panel_4, panel_5,
        label_1, label_2, label_3, label_4, label_5,
        quit_menu_item, text_menu_item_2,
        check_box_1, check_box_2, check_box_3,
        table_column_1, table_column_2, check_box_group

    #
    # Initialize some data for the objects.
    #
    oses := ["Windows", "Linux", "Solaris"]
languages := ["C", "C++", "Java", "Icon"]
shares := [["Microsoft", "101.84"], ["Oracle", "32.52"], ["IBM", "13.22"],
           ["Intel", "142.00"]]

    #
    # Set the attributes, then setup a simple menu system
    #
    attrib("size=490,400", "min_size=490,400", "font=sans",
         "bg=light gray", "label=Second example", "resize=on")
    menu_bar := MenuBar()
    menu := Menu("label=File")
    quit_menu_item := TextMenuItem("label=Quit")
    quit_menu_item.connect(self, "handle_quit", ACTION_EVENT)
    menu.add(quit_menu_item)
    text_menu_item_2 := TextMenuItem("label=Message")
18.3. A MORE COMPLEX DIALOG EXAMPLE

text_menu_item_2.connect(self, "handle_text_menu_item_2",
    ACTION_EVENT)
menu.add(text_menu_item_2)
menu_bar.add(menu)
add(menu_bar)

# # Set-up the checkbox panel #
check_box_group := CheckBoxGroup()
panel_1 := Panel("pos=20,50", "size=130,130")
label_2 := Label("pos=0,0", "internal_alignment=l", "label=Favorite OS")
panel_1.add(label_2)
check_box_1 := CheckBox("pos=0,30")
check_box_1.set_label(oses[1])
check_box_1.connect(self, "handle_check_box_1", ACTION_EVENT)
check_box_group.add(check_box_1)
panel_1.add(check_box_1)
check_box_2 := CheckBox("pos=0,60")
check_box_2.set_label(oses[2])
check_box_group.add(check_box_2)
check_box_2.connect(self, "handle_check_box_2", ACTION_EVENT)
panel_1.add(check_box_2)
check_box_3 := CheckBox("pos=0,90")
check_box_3.set_label(oses[3])
check_box_group.add(check_box_3)
check_box_3.connect(self, "handle_check_box_3", ACTION_EVENT)
panel_1.add(check_box_3)
add(panel_1)

# # The text-list of messages. #
panel_2 := Panel("pos=220,50", "size=100%-240,50%-60")
label_1 := Label("pos=0,0", "internal_alignment=l", "label=Messages")
panel_2.add(label_1)
text_list := TextDisplay("pos=0,30", "size=100%,100%-30")
text_list.set_contents([])
panel_2.add(text_list)
add(panel_2)

# # The table of shares. #
panel_3 := Panel("pos=220,50","size=100%-240,50%-40")
table := Table("pos=0,30","size=100%,100%-30", "select_one")
table.connect(self, "handle_table", SELECTION_CHANGED_EVENT)
table.set_contents(shares)

table_column_1 := TableColumn("label=Company", "internal_alignment=l",
    "column_width=100")
table_column_1.connect(self, "handle_table_column_1", ACTION_EVENT)
table.add_column(table_column_1)
table_column_2 := TableColumn("label=Share price", "internal_alignment=r",
    "column_width=100")
table_column_2.connect(self, "handle_table_column_2", ACTION_EVENT)
table.add_column(table_column_2)
panel_3.add(table)
label_5 := Label("pos=0,0", "internal_alignment=l", "label=Shares")
panel_3.add(label_5)
add(panel_3)

# The drop-down list of languages.
#
panel_4 := Panel("pos=20,190", "size=180,50")
list := List("pos=0,30", "size=100,")
list.connect(self, "handle_list", SELECTION_CHANGED_EVENT)
list.set_selection_list(languages)
panel_4.add(list)
label_3 := Label("pos=0,0", "internal_alignment=l", "label=Favorite language")
panel_4.add(label_3)
add(panel_4)

# The text field.
#
panel_5 := Panel("pos=20,280", "size=180,50")
label_4 := Label("pos=0,0", "internal_alignment=l", "label=Enter a string")
panel_5.add(label_4)
text_field := TextField("pos=0,30", "size=130," "draw_border=t")
text_field.connect(self, "handle_text_field", TEXTFIELD_CHANGED_EVENT)
panel_5.add(text_field)
add(panel_5)
end

# Simple main procedure just creates the dialog.
#
procedure main()
    SecondTest().show_modal()
end
18.3. A MORE COMPLEX DIALOG EXAMPLE

Start examining this program with the `component_setup()` method at the end. This method initializes some data and sets attributes. This includes several icon graphics attributes, as well as the minimum size, which is an attribute of the `Dialog` class.

The next part creates a menu bar structure. Menu structures are presented in detail later in this chapter, but for now observe that this code creates two textual menu items within a `Menu` object, which is itself within a `MenuBar` object, which is added to the dialog. Both menu items are connected to event handler methods.

The next section sets up three check boxes, placed together with the label "Favorite OS" in a `Panel` object, which serves as a container for other objects that logically are treated as a whole. Objects within a `Panel` have their size and position computed relative to the `Panel` rather than the window. So, for example, the first `Label` object is positioned with "pos=0,0". This places it at the top left-hand corner of the `Panel`, not the top left of the window. Percentage specifications similarly relate to the enclosing `Panel`.

Each `CheckBox` is a separate object. To make the three check boxes behave as a group so that when one is checked another is unchecked, they are placed in a `CheckBoxGroup` object; this just "brackets" them together. When grouped together in this way the checkboxes are called radio buttons because they work like the tuner buttons found on old car radios. Note that each `CheckBox` is added to the `CheckBoxGroup` and the `Panel`.

The next section is a `Panel` that holds a label ("Messages") and a `TextList` object. In this case the object holds a list of message strings that scroll by like a terminal window. A `TextList` object can also be used for selecting one or more items from a list of strings and
there is an EditableTextList, which can be used for editing text.

The third panel contains a label ("Shares"), and a Table object, which is used for displaying tabular data. Note that class Table has nothing to do with Icon’s table data type. Adding a TableColumn object to the table sets up each column. A table column’s initial column width is specified with attribute column_width and the alignment of the column’s contents is set with attribute internal_alignment. Attribute select_one configures the table to allow one row to be highlighted at a time. The default is not to allow highlighting of rows; the other option is to allow several to be highlighted at once with select_many.

The next panel contains another label ("Favorite language") and a drop-down list of selections, created using the List class. The selections are set using the set_selection_list() method. The final panel contains a label ("Enter a string") and a TextField object, which is used to obtain entry of a string from the keyboard.

Several of the components are connected to event handlers in the class. Each handler method adds a line to the list of strings in the TextList by calling the put_line() method. The text list thus gives an idea of the events being produced by the toolkit. The exception is the menu item Quit, which exits the program.

This dialog overrides the handle_event() method that is invoked by the toolkit for any component (including dialogs) that may elect to handle an event. In this case, the dialog just prints out the Icon event code for the particular event. This method also checks for the Alt-q keyboard combination, which closes the dialog.

18.4 More About Event Handling

As shown above, components generate events when something of interest happens. For example, a button generates an ACTION_EVENT when it is pressed. Different components generate different events, but there are some basic events generated by all components:

- MOUSE_PRESS_EVENT a mouse press within the component’s region.
- MOUSE_DRAG_EVENT a mouse drag within the component’s region.
- MOUSE_RELEASE_EVENT a mouse release within the component’s region.

For any non-mouse events, the Dialog class fires an ICON_EVENT.

Event are passed to listeners in an Event object. This object contains three fields, with corresponding getter methods, as follows:

- get_source() Returns the component which fired the event.
- get_type() Returns the type code, eg ICON_EVENT
- get_param() Returns an arbitrary parameter depending on the type. In nearly all cases this is the original underlying Icon graphics event.
The `get_param()` method is necessary, for example, to distinguish between a left mouse click and a right mouse click on a `MOUSE_RELEASE_EVENT`; for instance

```plaintext
method on_release(ev)
    if ev.get_param() === &rrelease then
      ... process right mouse up
    end
```

18.5 Containers

Containers are components that contain other components. The `Dialog` class is a container, as is the `Panel` class seen in the last example. Two other useful container objects in the standard toolkit are `TabSet` and `OverlaySet`.

**TabSet**

This class contains several tabbed panes, any one of which is displayed at any given time. The user switches between panes by clicking on labeled tabs at the top of the object. The `TabSet` contains several `TabItems`, each of which contains the components for that particular pane. To illustrate this, Listing 18-3 presents a simple example of a `TabSet` that contains three `TabItems`, each of which contains a single label.

Listing 18-3

TabSet Program

```plaintext
import gui
#include "guih.icn"
#
# Simple example of a TabSet
#
class Tabs : Dialog(quit_button)
    method change(e)
        write("The tabset selection changed")
    end

    method component_setup()
        local tab_set, tab_item_1, tab_item_2, tab_item_3
        attrib("size=355,295", "font=sans", "bg=light gray",
                "label=TabSet example", "resize=on")
    #
    # Create the TabSet
```
tab_set := TabSet("pos=20,20", "size=100%-40,100%-80")
#
# First, second, and third panes
#
tab_item_1 := TabItem("label=Pane 1")
tab_item_1.add(Label("pos=50%,50%", "align=c,c", "label=Label 1"))
tab_set.add(tab_item_1)
tab_item_2 := TabItem("label=Pane 2")
tab_item_2.add(Label("pos=50%,50%", "align=c,c", "label=Label 2"))
tab_set.add(tab_item_2)
tab_item_3 := TabItem("label=Pane 3")
tab_item_3.add(Label("pos=50%,50%", "align=c,c", "label=Label 3"))
tab_set.add(tab_item_3)
tab_set.set_which_one(tab_item_1)
tab_set.connect(self, "change", SELECTION_CHANGED_EVENT)
add(tab_set)
#
# Add a quit button; close the dialog when it is pressed
quit_button := TextButton("pos=50%,100%-30", "align=c,c", "label=Quit")
quit_button.connect(self, "dispose", ACTION_EVENT)
add(quit_button)
connect(self, "dispose", CLOSE_BUTTON_EVENT)
end
end
procedure main()
    Tabs().show_modal()
end

The resulting window is shown in Figure 18-3:
One interesting point in this dialog is the line:

```python
connect(self, "dispose", CLOSE_BUTTON_EVENT)
```

A dialog generates an event whenever the close button is pressed. Connecting this event to the `dispose` method configures the dialog to close when this button is pressed.

### OverlaySet

An **OverlaySet** is like a **TabSet**, but the pane on display is under program control; there are no tabs to click. Instead of adding items to **TabItem** structures, **OverlayItem** objects are used. An empty **OverlayItem** can be used when the area should be blank. The current **OverlayItem** on display is set by `set_which_one(x)`, where `x` is the desired **OverlayItem**.

## 18.6 Menu Structures

The toolkit provides the standard building blocks required to create a menu system (see Table 18-1). Customized components can be added; this is discussed later.

### Table 18-1

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuBar</td>
<td>The menu area along the top of the window, containing one or more <strong>Menus</strong>.</td>
</tr>
<tr>
<td>MenuButton</td>
<td>A floating menu bar containing one <strong>Menu</strong>.</td>
</tr>
<tr>
<td>Menu</td>
<td>A drop down menu pane containing other <strong>Menus</strong> or menu components.</td>
</tr>
<tr>
<td>TextMenuItem</td>
<td>A textual menu item.</td>
</tr>
<tr>
<td>CheckBoxMenuItem</td>
<td>A checkbox in a menu which can be part of a <strong>CheckBoxGroup</strong> if desired.</td>
</tr>
<tr>
<td>MenuSeparator</td>
<td>A vertical separation line between items.</td>
</tr>
</tbody>
</table>

Items in a **Menu** can have left and right labels as well as customized left and right images.

To see how this all fits together, Listing 18-4 shows our next example program.

### Listing 18-4

A Menu Example Program

```python
import gui
$include "guih.icn"

class MenuDemo : Dialog()
```
method component_setup()
    local file_menu, menu_bar, check_box_group, text_menu_item,
    labels_menu, images_menu, checkboxes_menu, group_menu,
    alone_menu, menu_button, check_box_menu_item, button_menu

    attrib("size=426,270", "font=sans", "bg=light gray", "label=Menu example")
    check_box_group := CheckBoxGroup()

    # Create the menu bar. The position and size default to
    # give a bar covering the top of the window.
    # The first menu ("File") - just contains one text item.
    menu_bar := MenuBar()
    file_menu := Menu("label=File")
    text_menu_item := TextMenuItem("label=Quit")
    text_menu_item.connect(self, "dispose", ACTION_EVENT)
    file_menu.add(text_menu_item)
    menu_bar.add(file_menu)

    # The second menu ("Labels") - add some labels, followed by
    # a separator and another text item
    labels_menu := Menu("label=Labels")
    labels_menu.add(TextMenuItem("label=One"))
    labels_menu.add(TextMenuItem("label=Two", "label=left=ABC"))
    labels_menu.add(MenuSeparator())
    labels_menu.add(TextMenuItem("label=Three", "label=right=123"))
    # A sub-menu in this menu, labeled "Images", contains
    # three text items with custom images. The rather unwieldy
    # strings create a triangle, a circle and a rectangle.
    images_menu := Menu("label=Images")
    text_menu_item := TextMenuItem("label=One")
    text_menu_item.set_img_left("15,c1,0------
    ------0---------000------000----
    ------000------000------000-----
    ------000------000------000-----
    ------000------000------000-----
    ------000------000------000-----
    ------000------000------000-----")
    images_menu.add(text_menu_item)
    text_menu_item := TextMenuItem("label=Two")
    text_menu_item.set_img_left("15,c1,000-----
    -----000------000------000-----
    -----000------000------000-----
    -----000------000------000-----
    -----000------000------000-----")
    images_menu.add(text_menu_item)
18.6. MENU STRUCTURES

```
# The third menu ("Checkboxes")
# Sub-menu - "Group" - two checkboxes in a checkbox group.
#
checkboxes_menu := Menu("label=Checkboxes")
group_menu := Menu("label=Group")
check_box_menu_item := CheckBoxMenuItem("label=One")
group_menu.add(check_box_menu_item)
check_box_menu_item := CheckBoxMenuItem("label=Two")
group_menu.add(check_box_menu_item)
checkboxes_menu.add(group_menu)
#
# Sub-menu - "Alone" - two checkboxes on their own
#
alone_menu := Menu()
alone_menu.set_label("Alone")
check_box_menu_item_3 := CheckBoxMenuItem("label=Three")
alone_menu.add(check_box_menu_item_3)
check_box_menu_item_4 := CheckBoxMenuItem("label=Four")
alone_menu.add(check_box_menu_item_4)
checkboxes_menu.add(alone_menu)
menu_bar.add(checkboxes_menu)
#
# Finally, create a menu button - a mini floating menu with
# one menu inside it.
```
### 18.7 Trees

The toolkit contains a tree component, which can be used to represent hierarchical data. To use it, it is necessary to create a tree-like data structure of `Node` objects. Children are added to a `Node` using its `add()` method. For example:

```plaintext
# This is the menu, its label appears on the button. It just
# contains a couple of text items for illustration purposes.
#
button_menu := Menu("label=Click")
button_menu.add(TextMenuItem("label=One"))
button_menu.add(TextMenuItem("label=Two"))
menu_button.set_menu(button_menu)
add(menu_button)
end

procedure main()
    MenuDemo().show_modal()
end
```

The output of this program with the middle menu and its submenu open appears in Figure 18-4.

Figure 18-4: Menus
18.7. TREES

root := Node("label=Root")
child1 := Node("label=Child1")
child2 := Node("label=Child2")
root.add(child1)
root.add(child2) # ...etc

After setting up the tree of Nodes, the root is passed to the Tree for display:

tree := Tree("pos=0,0", "size=100,100") tree.set_root_node(root)

The tree data structure can change dynamically over time. When this occurs, the Tree must be notified of the change by invoking the tree_structure_changed() method.

The Tree class generates events when the selected Node (or Nodes) changes, and also when part of the tree is expanded or collapsed by the user.

The next example uses a Tree with a Table and a Sizer to provide a file system explorer program. The Sizer is a narrow area between the tree and the table which can be dragged to resize both dynamically. Because of the toolkit’s relatively simple layout mechanism, the resizing code in handle_sizer() is quite awkward.

Figure 18-5: Explorer

Listing 18-5

The Explorer Program
import gui

$include "keysyms.icn"
$include "guih.icn"

#
# A very simple filesystem explorer with a tree and a table.
#
class Explorer : Dialog(tree, sizer, tbl)

    #
    # Given a Node n, get the full file path it represents by
    # traversing up the tree structure to the root.
    #
    method get_full_path(n)
        local s := ""
        repeat {
            s := n.get_label() || s
            n := n.get_parent_node() | break
        }
        return s
    end

    #
    # Invoked when a sub-tree is expanded (ie: the little + is
    # clicked). An expansion event also includes contractions too
    #
    method handle_tree_expansion()
        local n := tree.get_last_expanded()
        #
        # Check whether it was an expansion or a contraction. If
        # an expansion, load the subtree and refresh the tree.
        #
        if n.isExpanded() then {
            load_subtree(n)
            tree.tree_structure_changed()
        }
    end

    #
    # Invoked when a row in the tree is selected (or de-selected).
    # If something is selected, load the table. We may not have something
    # selected if the user contracted the parent node of the selected node.
    #
method handle_tree_selection()
   local n
   if n := tree.object_get_selections()[1] then
      load_table(n)
   end

#
# Given a Node n, load its children with the sub-directories.
#
method load_subtree(n)
   local name, r1, dir_list := get_directory_list(get_full_path(n))
   n.clear_children()
   every name := !dir_list[1] do {
      if (name ~== "./" ) & (name ~== "./.") then {
         r1 := Node("always_expandable=t")
         r1.set_label(name)
         n.add(r1)
      }
   }
end

#
# Given a Node n, load the table with the sub-files and sub-directories.
#
method load_table(n)
   local s := get_full_path(n), l := [ ], t := get_directory_list(s)
   every el := !sort(t[1] ||| t[2]) do {
      p := stat(s || el) | stop("No stat")
      put(l, [el, p.size, ctime(p.mtime)[5:17], p.mode])
   }
   tbl.set_contents(l)
   tbl.goto_pos(1, 0)
end

#
# The sizer has moved, so reset the sizes and positions of the
# table, tree and sizer. Then call resize() to reposition
# everything.
#
method handle_sizer(ev)
   result_x := sizer.get_curr_pos()
   tree.set_size(result_x - 10, tree.h_spec)
   sizer.set_pos(result_x, sizer.y_spec)
chapter 18. object-oriented user interfaces

tbl.set_pos(result_x + 10, tbl.y_spec)
tbl.set_size("100%-" || string(result_x + 20), tbl.h_spec)
resize()
end

#
# Catch Alt-q to close the dialog.
#
method quit_check(ev)
  if ev.get_param() === "q" & meta then dispose()
end

#
# Override resize to set the sizer's min/max locations.
#
method resize()
  self.Dialog.resize()
  sizer.set_min_max(135, get_w_reference() - 160)
end

method component_setup()
  local root_node

  attrib("size=750,440", "resize=on", "label=Explorer")
  connect(self, "dispose", CLOSE_BUTTON_EVENT)
  connect(self, "quit_check", ICON_EVENT)
  tree := Tree("pos=10,10", "size=250,100%-20", "select_one")
  tree.connect(self, "handle_tree_expansion", TREE_NODE_EXPANSION_EVENT)
  tree.connect(self, "handle_tree_selection", SELECTION_CHANGED_EVENT)
  add(tree)

  tbl := Table("pos=270,10", "size=100%-280,100%-20", "select_none")
  tbl.add_column(TableColumn("label=File", "column_width=150"))
  tbl.add_column(TableColumn("label=Size", "column_width=75", "internal_alignment=r"))
  tbl.add_column(TableColumn("label=Date", "column_width=100"))
  tbl.add_column(TableColumn("label=Bits", "column_width=100"))
  add(tbl)

  sizer := Sizer("pos=260,10", "size=10,100%-20")
  sizer.connect(self, "handle_sizer", SIZER_RELEASED_EVENT)
  add(sizer)

  #
  # Initialize the tree data structure.
18.8. OTHER COMPONENTS

This section gives examples of some components that have not yet been encountered. For full details of how to use these classes, and the available methods and options, please see the GUI class library reference in Appendix C.

Borders

This class provides decorative borders. Optionally, a single other component can be the title of the Border. This would normally be a Label object, but it could also be a CheckBox or an Icon, or whatever is desired. The set_title(c) method is used to set the title. Here is a program fragment to create a border with a label as its title:

```plaintext
b := Border()
#
# Add a Label as the title
#
l := Label("label=Title String")
b.set_title(l)
add(b)
```

The Border class acts as a container (like a Panel), and so objects may be placed within it in the same way.

Images and icons

The toolkit supports both images loaded from files and bitmap icons defined by strings. Images stored in files are manipulated using the Image class. The method set_filename(x) specifies the location of the file to be displayed. The image is scaled down to the size of the object, and optionally may be scaled up if it is smaller. A border may be used if desired.
Icons are created using the `Icon` class. The icon string is set using the `set_img()` method; again a border can be used if desired. Finally, the `IconButton` class lets icons serve as buttons, producing events when they are clicked.

**Scroll bars**

Horizontal and vertical scroll bars are available with the `ScrollBar` class. Scroll bars can be used either in the conventional way, in which the button in the bar represents a page size, and the whole bar represents a total, or as a slider in which case the button simply moves over a specified range of numbers.

### 18.9 Custom Components

This section looks at how you can create customized components that can be added to dialogs. You might want to do this to have circular rather than rectangular buttons in your dialog, for example. Or perhaps you might want to add a substantial new component type in an application, such as the main grid area in a spreadsheet program.

#### Creating new components

Every component is a subclass of the `Component` class. This class contains several methods and variables that can be used by a new component. Custom components will extend this class by implementing its abstract class `display()`, and very possibly overriding and replacing one or more of `Component`’s methods, such as `resize()`.

A full list of the methods and variables defined in the `Component` class is given in the GUI class library reference in Appendix C. Please refer to that section when reading this next example, which comes from the `gui` package itself: a simple "progress bar" component.

Listing 18-6

A ProgressBar Component

```java
package gui

#include "guih.icn"

#
# A progress bar
#
class ProgressBar : Component(
    p, # The percentage on display.
    bar_x, bar_y,
    bar_h, bar_w) # Maximum bar height and width.

    method resize()
```
# Set a default height based on the font size.
/h_spec := WAttrib(cwin, "fheight") + 2 * DEFAULT_TEXT_Y_SURROUND
#
# Call the parent class's resize method (this is mandatory).
self.Component.resize()
#
# Set bar height and width figures - this just gives a
# sensible border between the "bar" and the border of the
# object. By using these constants, a consistent
# appearance with other objects is obtained.
bar_x := x + DEFAULT_TEXT_X_SURROUND
bar_y := y + BORDER_WIDTH + 3
bar_w := w - 2 * DEFAULT_TEXT_X_SURROUND
bar_h := h - 2 * (BORDER_WIDTH + 3)
end

method display(buffer_flag)
#
# Erase and re-draw the border and bar
EraseRectangle(cbwin, x, y, w, h)
DrawRaisedRectangle(cbwin, x, y, w, h)
FillRectangle(cbwin, bar_x, bar_y, bar_w * p / 100.0, bar_h)
#
# Draw the string in reverse mode
cw := Clone(cbwin, "drawop=reverse")
center_string(cw, x + w / 2, y + h / 2, p || "%")
Uncouple(cw)
#
# Copy from buffer to window if flag not set.
#
if /buffer_flag then CopyArea(cbwin, cwin, x, y, w, h, x, y)
end
#
# Get the current percentage.
#
method get_percentage()
  return p
end
#
# Set the percentage.
#
method set_percentage(p)
    p <=: 0
    p >=: 100
    self.p := p
    invalidate()
end

#  
# Provide an extra attribute, "percentage"
#  
method set_one(attr, val)
    case attr of {
        "percentage": set_percentage(int_val(attr, val))
        default: self.Component.set_one(attr, val)
    }
end

initially(a[])
    self.Component.initially()
    set_percentage(0)
    set_fields(a)
end

An example of a ProgressBar in use is shown in Figure 18-6.

Figure 18-6: Progress bar in use

More complex components use other components within themselves. For example, the TextList class contains two ScrollBars, each of which in turn contains two IconButton.s. The Component class has support for contained objects built in, so creating components of this sort is quite easy.

The **ProgressBar** in use is shown in Figure 18-6.

Figure 18-6: Progress bar in use

More complex components use other components within themselves. For example, the **TextList** class contains two **ScrollBar**s, each of which in turn contains two **IconButton**s. The **Component** class has support for contained objects built in, so creating components of this sort is quite easy.
Figure 18-7 shows a dialog window containing another example component which contains an IconButton and a Label as subcomponents. A list of strings is given as an input parameter. When the button is pressed the label changes to the next item in the list, or goes back to the first one. The user can thus select any of the items.

![Figure 18-7: Circulate component in use](image)

In this example, sub-components b and l are initialized in the component’s constructor, and added to the component using the `add` method. This ensures they are properly initialized. A listener is connected to the button so that the selection moves on one when it is pressed. The `set_selection()` method is available to a client program to change the selection programatically. Note how this method invokes `invalidate()`. This tells the toolkit that the component needs to be re-displayed. To keep the GUI responsive, the toolkit will only schedule this re-display when there are no user input events to be processed. So, `invalidate()` doesn’t actually invoke the component’s `display()` method directly.

The `resize()` method sets the sub-components’ position and size, and then calls the `resize()` method for each of them. The `display()` method erases the component’s rectangular area, draws a border, and draws the two sub-components into the area. The `set_one()` method is also overridden to provide some custom attributes for the component. For all other attributes, the parent class’s `set_one()` method is invoked, meaning all the standard attributes work too. So, a client can construct a `Circulate` with something like:

```plaintext
c := Circulate("size=250", "pos=20,20", "selection_list=hot,warm,cold", "selection=2", "bg=green")
```

Note how the height is omitted; the `resize()` method will set a default value.

Listing 18-7
Circulate component
package gui
link graphics

$include "guih.icn"

# # Selection from a list #
class Circulate : Component (selection, selection_list, b, l)
  # # Set the list from which selections are made.
  # # @param x the list of selection strings
  method set_selection_list(x)
    selection_list := x
    set_selection(1)
    return x
  end

  # # Set the selection to the given index into the selection list.
  # # @param x an index into the selection list
  method set_selection(x)
    selection := x
    l.set_label(selection_list[x])
    invalidate()
    return x
  end

  # # Return the current selection, as an index in the selection list.
  # # @return an integer, being the current selection
  method get_selection()
    return selection
  end

  # # Called once at startup, and whenever the window is resized.
  # method resize()
  /h_spec := WAttrib(cwin, "fheight") + 16
  compute_absolutes()
# Set button position and size
b.set_pos(BORDER_WIDTH, BORDER_WIDTH)
b.set_size(h - 2 * BORDER_WIDTH, h - 2 * BORDER_WIDTH)
b.resize()

l.set_pos(h - BORDER_WIDTH + DEFAULT_TEXT_X_SURROUND, h / 2)
l.set_align("l", "c")
l.set_size(w - h - 2 * DEFAULT_TEXT_X_SURROUND,
          h - 2 * BORDER_WIDTH)
l.resize()
return
end

# Display the object. In this case, double buffering is not necessary.
#
method display(buffer_flag)
  W := if /buffer_flag then cwin else cbwin
  EraseRectangle(W, x, y, w, h)
  DrawSunkenRectangle(W, x, y, w, h)
  l.display(buffer_flag)
  b.display(buffer_flag)
  do_shading(W)
end

# The handler for the button - move the selection forward.
#
method on_button_pressed(ev)
  set_selection(1 + selection % *selection_list)
  create_event_and_fire(SELECTION_CHANGED_EVENT, e)
end

method set_one(attr, val)
  case attr of {
    "selection" : set_selection(int_val(attr, val))
    "selection_list" : set_selection_list(val)
    default: self.Component.set_one(attr, val)
  }
end

initially(a[])

CHAPTER 18. OBJECT-ORIENTED USER INTERFACES

self.Component.initially()
l := Label()
l.clear_draw_border()
add(l)
b := IconButton()
b.connect(self, "on_button_pressed", ACTION_EVENT)
add(b)
b.set_draw_border()
b.set_img("13,c1,~~0000~~
~000000~~00~~00~~00~~00~~00~~
`00~~00~~00~~00~~00~~00~~00~~
`00~~000000~~00~~00~~00~~00~~
`00~~00~~00~~00~~00~~00~~00~~
~~0000~~")
set_fields(a)
end

Customized menu components

Listing 18-8 contains a custom menu component. The class hierarchy for menu structures is different to other components, and so this component is a subclass of SubMenu, rather than Component. The component allows the user to select one of a number of colors from a Palette by clicking on the desired box. Again, please read this example in conjunction with the reference section on menus.

Listing 18-8
Color Palette Program

import gui

#
# Include the standard constants. Define width of one colour cell in pixels.
#
$include "guih.icn"
$define CELL_WIDTH 30

class Palette : SubMenu(
    w, h,       # width and height
colour,      # Color number selected
palette,     # List of colors
box_size,    # Width/height in cells
temp_win     # Temporary window
)
# Get the result
#
method get_colour()
    return palette[colour]
end
#
# Set the palette list
#
method set_palette(l)
    box_size := integer(sqrt(*l))
    return palette := l
end
#
# This is called by the toolkit; it is a convenient place to initialize sizes.
#
method resize()
    w := h := box_size * CELL_WIDTH + 2 * BORDER_WIDTH
end
#
# Called to display the item. The x, y co-ordinates have been set up
# for us and give the top left hand corner of the display.
#
method display()
    if /temp_win then {
        # Open a temporary area for the menu and copy.
        temp_win := WOpen("canvas=hidden", "size=\" || w || \", || h)
        CopyArea(parent_component.get_parent_win(),
            temp_win, x, y, w, h, 0, 0)
    }

cw := Clone(parent_component.cwin)
#
# Clear area and draw rectangle around whole, then draw the color grid
EraseRectangle(cw, x, y, w, h)
DrawRaisedRectangle(cw, x, y, w, h)
y1 := y + BORDER_WIDTH
e := create "fg=" || lpalette
every 1 to box_size do {
    x1 := x + BORDER_WIDTH

every 1 to box_size do {
    WAttrib(cw, @e)
    FillRectangle(cw, x1, y1, CELL_WIDTH, CELL_WIDTH)
    x1 +:= CELL_WIDTH
}
    y1 +:= CELL_WIDTH
}
Uncouple(cw)
end

#
# Test whether pointer in palette_region, and if so which cell it's in
#
method in_palette_region()
    if (x <= &x < x + w) & (y <= &y < y + h) then {
        x1 := (&x - x - BORDER_WIDTH) / CELL_WIDTH
        y1 := (&y - y - BORDER_WIDTH) / CELL_WIDTH
        return 1 + x1 + y1 * box_size
    }
end

#
# Will be called if our menu is open.
#
method handle_event(e)
    if i := in_palette_region() then {
        if integer(e) = (&lrelease | &rrelease | &mrelease) then {
            colour := i
            # This is a helper method in the superclass which
            # closes the menu system and fires an ACTION_EVENT
            succeed(e)
        }
    } else {
        if integer(e) = (&lrelease | &rrelease | &mrelease |
            &lpress | &rpress | &mpress) then
            # This is a helper method in the superclass which
            # closes the menu system, without firing an event.
            close_all()
    }
end
# Close this menu. Restore window area.
#
method hide()
    EraseRectangle(parent_component.cwin, x, y, w, h)
    CopyArea(temp_win, parent_component.get_parent_win(), 0, 0, w, h, x, y)
    WCLOSE(temp_win)
    temp_win := &null
end

# Support a "palette" attrib
method set_one(attr, val)
    case attr of {
        "palette" : set_palette(val)
        default : self.MenuComponent.set_one(attr, val)
    }
end

initially(a[])
    self.SubMenu.initially()
    #
    # Set the image to appear on the Menu above ours. We could design a tiny
    # icon and use that instead of the standard arrow if we wished.
    # Call set_fields to support the attrib style constructor.
    #
    set_img_right(img_style("arrow_right"))
    set_fields(a)
end

#
# Test class dialog.
#
class TestPalette : Dialog(palette)
    method on_palette(ev)
        write("Colour selected : \\ " || palette.get_colour())
    end

    method on_anything(ev)
        write("Anything item selected")
    end

    method component_setup()
        local menu_bar, menu, text_menu_item, close
        attrib("size=400,200")
# Create a MenuBar which includes our palette as a sub-menu
menu_bar := MenuBar("pos=0,0")
menu := Menu("label=Test")
text_menu_item := TextMenuItem("label=Anything")
text_menu_item.connect(self, "on_anything", ACTION_EVENT)
menu.add(text_menu_item)

palette := Palette("label=Test menu",
                   "palette=red,green,yellow,black,white,purple,gray,blue,pink")
palette.connect(self, "on_palette", ACTION_EVENT)
menu.add(palette)

add(menu_bar)

# Add a close button.
close := TextButton("pos=50%,66%", "align=c,c", "label=Close")
close.connect(self, "dispose", ACTION_EVENT)
add(close)
end

procedure main()
    TestPalette().show_modal()
end

The resulting window, with the **Palette** menu active is shown in Figure 18-8.

Figure 18-8: TestPalette window
18.9. **CUSTOM COMPONENTS**

**Tickers**

GUI programs spend most of their time waiting for user input events. The toolkit allows this spare time to be exploited by user programs by the use of "tickers". These are classes which subclass `Ticker`, and implement a single method, `tick()`, which is invoked at a specified interval. If the toolkit is busy processing events or updating the display, the actual interval may be much greater than that requested, but it will never be less.

Many components make use of tickers. For example when the mouse button is dragged below the bottom of an `EditableTextList` and held, the cursor scrolls downwards without any user events occurring. This is handled with a simple ticker:

```python
class FirstTicker : Ticker():
    method tick():
        write("Doing something in FirstTicker")
    end
end
```

If we had an instance `t` of this ticker, we would start it with:

```python
t.setTicker(1000)
```

and stop it with

```python
t.stopTicker()
```

Both calls would have to be made whilst a dialog was open, because it is from within the toolkit’s event processing loop that tickers are scheduled.

For convenience, the base `Component` class is a subclass of `Ticker`. This means that a dialog class (which is itself a subclass of `Component`, and hence of `Ticker` too), can simply implement a `tick()` method to use the ticker facility. It simply needs to invoke `setTicker()` and `stopTicker()` to start and stop the ticker, respectively. Another method, `retimeTicker()`, allows the rate of an active ticker to be changed.

One important rule regarding ticker programming has to be borne in mind, and that is that the `tick()` method implementation must return quite quickly; at most within a few tenths of a second. If it does not, then the GUI may become unresponsive to user events, which cannot be processed whilst control is in the `tick()` method.

The `tick()` method must return quickly. If the task you want to implement in the background is by nature deeply structured and takes a long time, it may be difficult to get out of the `tick()` method promptly and continue in the same state on the next tick. Co-expressions can be helpful here. A co-expression maintains its own stack and can suspend itself, and then continue again later with the state (including stack) intact.

Here is an example program which illustrates these points. It generates prime numbers using the Erastothenes’ sieve method, in a ticker, using a co-expression to conveniently
suspend generation after each prime. This program also introduces a new component, a slider which is used to increase or decrease the ticker rate dynamically.

Listing 18-9
Erastothenes’ Sieve Program

```
import gui
$include "guih.icn"
$define PRIME_LIMIT 20000

# A program to calculate prime numbers in a background ticker, and display them in a dialog.

class Sieve : Dialog(prime_ce, interval, count_label, prime_label,
    rate_label, start, stop)

    # Bring the label and ticker into line with the interval slider. If the ticker is running, retime it.
    method synch_interval()
        rate_label.set_label(interval.get_value() || " ms")
        if is_ticking() then retime_ticker(interval.get_value())
    end

    # Toggle the grey state of the start/stop buttons.
    method toggle_buttons()
        start.toggle_is_shaded()
        stop.toggle_is_shaded()
    end

    # When the start button is pressed, toggle the grey state and start the ticker.
    method on_start()
        toggle_buttons()
        set_ticker(interval.get_value())
    end

    # When the stop button is pressed, toggle the grey state and stop the ticker.
    method on_stop()
        toggle_buttons()
        stop_ticker()
    end
```
method on_stop()
    toggle_buttons()
    stop_ticker()
end

# The tick method, which is invoked regularly by the toolkit.
# It just invokes the co-expression to display the next prime.
#
method tick()
    @prime_ce
end

# This method constitutes the co-expression body.
#
method primes()
    local prime_candidate, non_prime_set := set(), prime_count := 0

    every prime_candidate := 2 to PRIME_LIMIT do {
        if not member(non_prime_set, prime_candidate) then {
            # Update the UI.
            count_label.set_label(prime_count +:= 1)
            prime_label.set_label(prime_candidate)
            # Update the non-prime set.
            every insert(non_prime_set,
                2 * prime_candidate to PRIME_LIMIT by prime_candidate)
            # Suspend the co-expression until the next tick.
            @&source
        }
    }
end

method component_setup()
    local prime_border, rate, buttons, b

    attrib("size=325,200", "label=Sieve")
    connect(self, "dispose", CLOSE_BUTTON_EVENT)

    prime_border := Border("pos=20,20", "size=100%-40,78")
    prime_border.set_title(Label("pos=10,0", "label=Primes"))
prime_ce := create primes()

prime_border.add(Label("pos=20,18", "label=Prime:"))
count_label := Label("pos=77,18", "size=40")
count_label.set_label('')
prime_border.add(count_label)
prime_border.add(Label("pos=20,40", "label=Value:"))
prime_label := Label("pos=77,40", "size=40")
prime_label.set_label('')
prime_border.add(prime_label)

add(prime_border)

rate := Panel("pos=20,112", "size=100%-40,30")
rate.add(Label("pos=0,50%", "size=45", "align=l,c", "label=Rate:"))
interval := Slider("pos=45,50%", "size=100%-90", "align=l,c", "range=20,2020", "is_horizontal=t")
interval.set_value(1000)
interval.connect(self, "synch_interval", SLIDER_DRAGGED_EVENT)
rate.add(interval)

rate_label := Label("pos=100%,50%", "size=45", "align=r,c", "internal_alignment=r")
rate.add(rate_label)
synch_interval()
add(rate)

buttons := Panel("pos=50%,158", "size=161,25", "align=c,t")
start := TextButton("pos=0,0", "label=Start")
start.connect(self, "on_start", ACTION_EVENT)
buttons.add(start)
stop := TextButton("pos=58,0", "label=Stop", "is_shaded=t")
stop.connect(self, "on_stop", ACTION_EVENT)
buttons.add(stop)
b := TextButton("pos=108,0", "label=Quit")
b.connect(self, "dispose", ACTION_EVENT)
buttons.add(b)
add(buttons)
end

procedure main()
    Sieve().show_modal()
end
18.10 Advanced List Handling

Several of the more sophisticated components extend a common base class, `SelectableScrollArea`, namely `TextList`, `Table` and `Tree`. (In fact, `Table` doesn’t directly extend `SelectableScrollArea`, it contains a header component and a content component that does). It is quite easy to add some advanced features to these components, such as right-click popup menus, multi-selection and drag and drop, and this is explained in this section.

**Selection**

Selection handling is straightforward. First, configure the component so that it allows selection of no, one, or many rows using the methods `set_select_none()`, `set_select_one()` or `set_select_many()`, or the attributes "select_none", "select_one" or "select_many". Then, listen for changes by listening for a `SELECTION_CHANGED_EVENT` to be fired:

```python
comp.connect(self, "handle_selection", SELECTION_CHANGED_EVENT)
```

When such an event does occur, the current selections can be retrieved in one of two ways. Either by getting the indexes of the selections using `get_selections()`, or by getting the objects selected, using `object_get_selections()`. The former returns a list of integers, the latter a list of objects whose type depends on the component. For a `TextList`, a list of strings is returned, for a `Table`, a list of lists (each being a row’s data), and for a `Tree`, a list of `Node` objects is returned. There are corresponding setter methods for setting the selection dynamically.
Popups

Adding popup menus is also easy. First create a Popup component, ready to be shown. Then, listen for a MOUSE_RELEASE_EVENT. Finally, when an event occurs check that is a right mouse release, and that the object is in the state you want. If it is, just activate the popup via its popup() method.

Drag and drop

The toolkit supports a limited form of drag and drop, which works only between components within the same window. To implement drag and drop, a class, DndHandler, must be subclassed and an instance "plugged-in" to the component which is a source or target of a potential drag and drop operation, using its set_dnd_handler() method. The DndHandler class provides five callback methods which the toolkit uses to control a drag and drop operation. When using the SelectableScrollArea family of components, it is best to subclass SelectableScrollAreaDndHandler, which is a custom subclass of DndHandler, with several methods already defined appropriately.

All of the above features are brought together in the following example program which provides a Tree and a TextList. Drag and drop is enabled between the two components, and both provide popup menus for adding/deleting elements.

Figure 18-10: Drag and drop

Listing 18-10 The Drag and drop Program
import gui
$include "guih.icn"

# # A DndHandler for the list #
class ListDndHandler : SelectableScrollAreaDndHandler()
   # # A drop has occurred; we succeed iff we accept it #
   method can_drop(d)
      local l, ll
      if l := parent.get_highlight() then {
         if d.get_source() === parent then # Move within the list itself
            parent.move_rows(parent.get_gesture_selections(), l)
         else {
            # # Copy from tree to list. d.get_content() gives # a list of the nodes being dragged.
            ll := []
            every el := !d.get_content() do # Don’t drag folders.
               if /el.is_folder_flag then put(ll, el.get_label())
               parent.insert_rows(ll, l)
         }
      return
   }
end

# # This is invoked after a successful operation when the # list was the source. If the destination (c) wasn’t the # list, then we must delete the rows from the list. #
method end_drag(d, c)
   if c ˜=== parent then
      parent.delete_rows(parent.get_gesture_selections())
   end
end

# # A DndHandler for the tree #
class TreeDndHandler : SelectableScrollAreaDndHandler()
   #
# Called during a drag event. We succeed iff the user is dragging over
# a row (this is handled by the parent) AND the thing we’re over is a folder.
#
method drag_event(d)
  if SelectableScrollAreaDndHandler.drag_event(d) then
    return 
      (parent.object_get_highlight().is_folder_flag)
  end
#
# A drop has occurred; we succeed iff we accept it.
# Only consider a drop on a folder.
#
method can_drop(d)
  local s, other, n, el
  if other := parent.object_get_highlight() & other.is_folder_flag then {
    if d.get_source() === parent then {
      #
      # If parent is the drop source, then we have a dnd from within
      # the tree. So, we just move the nodes.
      # d.get_content() will be a list of the nodes that were dragged.
      every el := !d.get_content() do {
        if el.get_parent_node().delete_node(el) then
          other.add(el)
      }
    } else {
      #
      # Drop from list. In this case d.get_content() will be a list of strings.
      every el := !d.get_content() do {
        n := TreeNode()
        n.set_label(el)
        other.add(n)
      }
    }
  } else {
    #
    # Notify the tree that the node data structure has altered.
    parent.tree_structure_changed()
    return
  }
end
#
# This is invoked after a successful operation when the tree was the source.
# If the destination (c) wasn’t the tree, we must delete the nodes from the tree.
#
method end_drag(d, c)
    if c !=== parent then {
        #
        # Delete all the nodes which will have been dragged.
        every n := !parent.object_get_gesture_selections() do
            if /n.is_folder_flag then
                n.get_parent_node().delete_node(n)
        #
        # Notify the tree that the node data structure has altered.
        parent.tree_structure_changed()
    }
end
end
#
# We use a custom Node subclass to also store an "is_folder_flag" flag.
#
class TreeNode : Node(is_folder_flag)
initially
    self.Node.initially()
    if \is_folder_flag then
        set_bmps([img_style("closed_folder"),
            img_style("closed_folder"), img_style("closed_folder")])
end
#
# The main dialog.
#
class DNDTest : Dialog(tree, lst, tree_popup, list_popup, new_folder_menu_item,
    delete_node_menu_item, delete_rows_menu_item)
#
# Delete nodes handler
#
method on_delete_node()
    local n, i, l

every n := !tree.object_get_gesture_selections() do
    n.get_parent_node().delete_node(n)
end
    tree.tree_structure_changed() # Notify tree of the change.
end
#

# Create a new folder
#
method on_new_folder()
    local n, o

    # Simply add a new node under the cursor, and notify the
    # tree that the data structure changed.
    #
    if o := tree.object_get_cursor() then {
        n := TreeNode(1)
        n.set_label("New folder")
        o.add(n)
        tree.tree_structure_changed()
    }
end

# Delete rows from the list
#
method on_delete_rows()
    lst.delete_rows(lst.get_gesture_selections())
end

# Add some rows to the list, at the cursor position, or at
# the top if there is no cursor.
#
method on_new_rows()
    lst.insert_rows(["new1", "new2", "new3"], lst.get_cursor() | 1)
end

# Helper method to create a tree structure.
#
method create_tree()
    local r := TreeNode(1), n
    r.set_label("root")

    every s := ("red" | "green" | "blue" | "yellow") do {
        n := TreeNode(1)
        n.set_label(s)
        r.add(n)
        every t := 1 to 5 do {

18.10. ADVANCED LIST HANDLING

```
o := TreeNode()
o.set_label(s || "-" || t)
n.add(o)
}
return r
end

# A selection-up event on the tree
#
method on_tree_release(ev)
  local n
  #
  # If the Icon event was a right mouse release, display the popup at the cursor.
  if ev.get_param() === &rrelease then {
    n := tree.object_get_cursor() | fail
    #
    # Adjust the shading depending on the node type.
    if /n.is_folder_flag then
      new_folder_menu_item.set_is_shaded()
    else
      new_folder_menu_item.clear_is_shaded()
    if n === tree.get_root_node() then
      delete_node_menu_item.set_is_shaded()
    else
      delete_node_menu_item.clear_is_shaded()
    tree_popup.popup()
  }
end

# A mouse release event on the list
#
method on_list_release(ev)
  if ev.get_param() === &rrelease then {
    #
    # If some rows to delete...
    if lst.get_gesture_selections() then
      delete_rows_menu_item.clear_is_shaded()
    else
      delete_rows_menu_item.set_is_shaded()
    list_popup.popup()
  }
```

method component_setup()
    local m, quit, mi
    attrib("size=350,295", "resize=on")
    connect(self, "dispose", CLOSE_BUTTON_EVENT)
    tree := Tree("pos=50%-10,10", "size=50%-20,100%-70",
        "align=r,t", "select_many", "show_root_handles=f")
    tree.set_root_node(create_tree())
    tree.set_dnd_handler(TreeDndHandler(tree))
    tree.connect(self, "on_tree_release", MOUSE_RELEASE_EVENT)
    add(tree)
    quit := TextButton("pos=50%,100%-40", "align=c,t", "label=Quit")
    quit.connect(self, "dispose", ACTION_EVENT)
    add(quit)
    # Create a TextList, with some arbitrary content.
    lst := TextList("pos=50%+10,10", "size=50%-20,100%-70", "select_many",
        "contents=one,two,three,four,five,six,seven,eight,nine,ten,eleven,_
        twelve,thirteen,fourteen,fifteen,sixteen,red,blue,green")
    lst.connect(self, "on_list_release", MOUSE_RELEASE_EVENT)
    lst.set_dnd_handler(ListDndHandler(lst))
    add(lst)
    tree_popup := PopupMenu()
    m := Menu()
    tree_popup.set_menu(m)
    delete_node_menu_item := TextMenuItem("label=Delete")
    delete_node_menu_item.connect(self, "on_delete_node", ACTION_EVENT)
    m.add(delete_node_menu_item)
    new_folder_menu_item := TextMenuItem("label=New folder")
    new_folder_menu_item.connect(self, "on_new_folder", ACTION_EVENT)
    m.add(new_folder_menu_item)
    add(tree_popup)
    list_popup := PopupMenu()
    m := Menu()
    list_popup.set_menu(m)
    delete_rows_menu_item := TextMenuItem("label=Delete")
    delete_rows_menu_item.connect(self, "on_delete_rows", ACTION_EVENT)
    m.add(delete_rows_menu_item)
    mi := TextMenuItem("label=Insert rows")
    mi.connect(self, "on_new_rows", ACTION_EVENT)
    m.add(mi)
    add(list_popup)
end
procedure main()
    DNDTest().show_modal()
end

18.11 Programming Techniques

Some of the earlier example dialogs were effectively “application windows.” In other words, the top-level window of a program. This section looks at some techniques for integrating dialog windows that are secondary or helper windows into a program.

Parameters

A dialog window will normally have parameters that the calling program will want to pass to it before it is displayed using the show() method. Possibly the attribute syntax "key=val" should be supported, and perhaps a default value should be set. All of these things are easily supported by following the following structure:

class AnyDialog : Dialog(a_variable)
    method set_a_variable(x)
        a_variable := x
    end
    ...
    method set_one(attr, val)
        case attr of {
            "a_variable" :
                set_a_variable(string_val(attr, val))
            default: self.Dialog.set_one(attr, val)
        }
    end

    method component_setup()
        # Initialize Components, possibly depending upon the value of a_variable
        ...
        # Configure the window itself...
        attrib("size=300,200")
    end

    initially(a[])
        self.Dialog.initially()
        a_variable := "default value"
        set_fields(a)
You then use the following code in the calling program:

```pascal
  d := AnyDialog()
  d.set_a_variable("something")
```
or

```pascal
  d := AnyDialog("a_variable=something")
```
or just

```pascal
  d := AnyDialog()
```
to use the default for `a_variable`. Furthermore, the standard dialog attributes can still be used as you would expect:

```pascal
  d := AnyDialog("a_variable=something", "font=times", "bg=green", "fg=red")
```

Subclassing can follow the same pattern. For example:

```pascal
class AnotherDialog : AnyDialog(another_variable)
  method set_another_variable(x)
    another_variable := x
  end
...
  method set_one(attr, val)
    case attr of {
      "another_variable" :
        set_another_variable(string_val(attr, val))
      default: self.AnyDialog.set_one(attr, val)
    }
  end

  method component_setup()
    self.AnyDialog.component_setup()
    ...
  end

initially(a[])
  self.AnyDialog.initially()
  another_variable := "default value"
  set_fields(a)
end
```
At first sight, it might seem that `set_fields()` will be invoked twice, which may cause problems. In fact, because we are calling the `AnyDialog` constructor with no parameters, the `set_fields()` call in that constructor has no effect. We just have to remember to call `set_fields(a)` in the `AnotherDialog` constructor itself. This will delegate its work up to the parent classes’ `set_one()` methods to handle all of the possible attributes we may give it.

Getting results out to the calling program is easy. The dialog can just set a result variable that can be retrieved by the caller using one of the dialog’s methods.

### 18.12 ivib

It can take many compiles and runs to get the components correctly sized and positioned in a dialog window with many components. Much of the code in a dialog is tedious to write. An interface builder called Ivib reduces the effort required for many common dialogs. Ivib allows a user to draw (interactively place and configure) components in a window area. Ivib’s saved files are program source code that implement the interface. Ivib was inspired by VIB, a program written by Mary Cameron and greatly extended by Gregg Townsend. The main window of Ivib, with a dialog under construction, is shown in Figure 18-11.

![Ivib main window](image)

**Figure 18-11: Ivib main window**

To create a dialog window using Ivib, start the program with the name of a new source file. For example:

```
ivib myprog.icn
```
The Ivib window will appear with a blank "canvas" area, which represents the dialog window to be created. At startup, the attributes of this window are the default Icon window attributes. Before you learn how to change these attributes, here is how you add a button to the dialog. Clicking the button in the top left-hand corner of the toolbar does this. Try moving and resizing the resulting button by left-clicking on it with the mouse. To change the label in the button, click on it so that the red borders appear in the edges. Then press Alt-D. The dialog shown in Figure 18-12 (left) appears. Change the label by simply changing the text in the "Label" field and clicking "Okay".

![Figure 18-12 Button (left) and Dialog (right) configuration windows](image)

As just mentioned, the dialog’s attributes are initially the default window attributes. To change these, select the menu option Canvas -> Dialog prefs. The window shown in Figure 18-12 (right) appears. To change the dialog attributes:

1. Click on the Attribs tab
2. Click on the Add button
3. Edit the two text fields so that they hold the attribute name and the attribute value respectively; for example try adding "bg" and "pale blue".
4. Click on Apply.
5. Click on Okay.

Note that the button changes its background to pale blue too. Each object has its own attributes that it can set to override the dialog attributes. Click on the button and press Alt-D to bring up the button’s configuration dialog again. Now click on the Attribs tab of
this dialog and set the background color to white, for example. Then click okay and you will see that the button’s background changes to white.

You will recall from the previous example programs that some objects can be contained in other objects, such as the Panel class. This is handled conveniently in Ivib. Add a Panel object to the dialog by clicking on the Panel button (on the second row, fourth from the left). A panel appears. Now drag the button into the panel. A message should appear in the information label below the toolbar, "Placed inside container." Now try dragging the panel about, and you will observe that the button moves too - it is now "inside" the panel. Dragging it outside the panel’s area moves it back out. This method applies to all the container objects.

There are several buttons that operate on objects. The large "X" button deletes the currently selected objects. Try selecting the button and deleting it. The arrow buttons are "redo" and "undo" operations. Clicking on the undo button will undo the delete operation and the button should reappear.

Now try saving your canvas. Press Alt-S, and select a filename, or accept the default. At the end of an ivib-enhanced Unicon source file is a gigantic comment containing ivib’s layout information. This comment is ASCII text, but it is not really human-readable. If the program is called, for example, myprog.icn, then this can be compiled with

```
unicon myprog
```

to give an executable file myprog that, when run, will produce the same dialog shown in the canvas area. Of course, the resulting dialog will not do anything, and it is then up to the programmer to fill in the blanks by editing myprog.icn.

Hopefully, following the above steps will give you an idea of how the Ivib program works. Below are more details regarding the individual components, dialogs, and operations.

### Moving, selecting and resizing

Select an object by clicking on it with the mouse. Its selection is indicated by red edges around the corners. Multi-select objects by holding the shift key down and clicking on the objects. The first selected object will have red corners; the others will have black corners. There are several functions which operate on multiple objects and map some attribute of the first selected object to the others; hence the distinction.

To move an object, select it by clicking on it with the left mouse button, and then drag it. Note that dragging an object even by one pixel will set the X or Y position to an absolute figure, disturbing any carefully set up percentage specification! Because this can be irritating when done accidentally, the X and/or Y position may be fixed in the dialog so that it cannot be moved in that plane. Alternatively, when selecting an object that you do not intend to move, use the right mouse button instead.

To resize an object, select it and then click on one of the corners. The mouse cursor will change to a resize cursor and the corner may be dragged. Note that, like the position
specification, resizing an object will set the size to an absolute number and will also reset the "use default" option. Again, this can be avoided by fixing the width and/or height in the object’s dialog box.

**Dialog configuration**

This dialog, accessed via the menu selection `Canvas -> Dialog prefs` allows the user to configure some general attributes of the dialog being created. The tabs are described in this section.

**Size** The minimum width and height entries simply set the minimum dimensions of the window. The width and height may be configured here, or more simply by resizing the canvas area by clicking on the red bottom right-hand corner.

**Attribs** This has been described briefly above; the Add button produces a new entry that is then edited. The edited entry is placed in the table with Apply. The Delete button deletes the currently highlighted selection.

**Code generation** The part of the code to setup the dialog is written into a method called setup. If the "interpose in existing file" option is checked, then the program will read the present contents of the selected output file up to the current setup method, interpose the new setup, and copy the remainder out. This is useful if some changes have been made to the file output by a previous run. It is important to take a copy of the existing file before using this option, in case unexpected results occur.

The other options simply select which other methods should be produced. If a main procedure is produced, then the result will be an executable program.

**Other** This tab allows the name of the dialog to be set, together with a flag indicating whether it is "modal" or not. If so, then a method called pending is produced. This method is repeatedly called by the toolkit while it is waiting for events to occur.

**Component configuration**

Each component dialog has a standard tabbed pane area for configuration of attributes common to all components. The tabs are as follows:

**Position & size** The X, Y, W and H options set the position and size of the object. The drop-down list can be used for convenience, or a value may be entered by hand. The "fix" buttons prevent the object from being moved or sized outside the given parameter in the Canvas area. This is useful once an object’s position has been finalized, and you don’t wish
to accidentally move it. The "use default" buttons mean that the width/height will be set
as the default for the object based on the parameters and the attributes. For example, a
button’s size will be based on the label and the font. For some objects there is no default
size, so these buttons are shaded. The alignment of the object is also set from this tab.

**Attribs**  This works in exactly the same way as the Attribs tab for the dialog, except that
these attributes apply only to the object.

**Other**  This tab allows the name used in the output program code to be set. The "Draw
Border" button applies to some objects (see the reference section in Appendix C for further
information). If the "Is Shaded" button is clicked, then the initial state of the object will
be shaded. If the "Has initial focus" button is clicked, then this object will have the initial
keyboard focus when the dialog is opened.

### Component details

Components are added to the dialog by clicking on the toolbar buttons, and dialogs are
produced by selecting the object and pressing Alt-D, as explained above. Most of the
dialogs are hopefully straightforward, but some warrant further explanation.

**TextButton**  The dialog for this component includes an option for the button to be added
to a **ButtonGroup** structure. This is explained in detail shortly.

**Border**  To select this component, rather than clicking inside the border, click in the area
at the bottom right hand corner. It can then be resized or moved. Now try dragging another
object, such as a **CheckBox** or **Label** and release it so that its top left-hand corner is within
the area in the bottom right-hand corner of the **Border** object. The **CheckBox/Label** or
whatever is now the title of the **Border**. Thus, any object can be in the title. To remove
the object from the **Border**, just drag it out. The alignment of the title object is set in the
dialog, but is by default left aligned.

**Image**  Initially the Image object has an outline. When a filename is entered into the
dialog however, the image itself is displayed.

**CheckBox**  Customized up/down images may be set from the dialog, and a **CheckBox-
Group** may be selected if one is available; this is explained in more detail shortly.

**MenuBar**  This dialog enables a complete multi-level menu to be created. Clicking on a
line allows an item to be inserted at that point. Only menus can be inserted into the lowest
level. Clicking on an item will allow insertion, deletion, or editing of the particular item.
A **CheckBoxGroup** can be created by selecting multiple check boxes (by holding down the Shift key while clicking on the lines) and clicking the button.

**ScrollBar** Both vertical and horizontal scroll bars are available; for details of how the various options work, please see the reference manual for the toolkit.

**Table** A table column is added by clicking the Add button; the details should then be edited and the Apply button pressed to transfer the details to the table. A selected column can be deleted with the Delete button. The drop-down list selects whether or not lines in the table can be selected by clicking them.

**TabSet** Add a **TabItem** (a tabbed pane) by clicking the Add button. A single pane is automatically present when the object is created. To switch between panes, select a **TabItem** button. An asterisk appears by the entry. When the dialog exits, this **TabItem** is on the front of the **TabSet**. To add items to the current pane, drag and drop them into it. The whole of the item must be in the pane, and a confirmation message appears to indicate that the item has been added to the container. To take it out of the container, drag it out of the pane. Note that the selected pane is the one that is configured to be initially at the front when the dialog is opened.

**MenuButton** The **MenuButton** component is a menu system with one root menu. The dialog is the same as **MenuBar**’s except that the small icon can be configured.

**OverlaySet** The configuration for an **OverlaySet** is very similar to that for a **TabSet**, except that there are no tab labels to configure of course.

**CheckBoxGroup** This does not create an object, it places several selected **CheckBox** objects into a **CheckBoxGroup** object. They act as coordinated radio buttons. To use this button, select several **CheckBox** objects, and press the button.

The **CheckBoxGroup** is configured by selecting the menu item Canvas -> CheckBoxes. The only attribute to be configured is the name of the **CheckBoxGroup**. Once a **CheckBoxGroup** has been created, it cannot be deleted. A **CheckBox** can be taken out or put into a **CheckBoxGroup** from its configuration dialog.

**ButtonGroup** The **ButtonGroup** operates in a very similar fashion to **CheckBoxGroup**, except that it places buttons into a **ButtonGroup**.
Other editing functions

Other editing functions can be applied to the dialog being created. They are accessed either via the toolbar, the Selection menu, or the Edit menu.

**Delete**  The Delete function simply deletes all the selected objects; note that deleting a container also deletes all the objects inside it.

**Undo and Redo**  The Undo and Redo functions undo and redo changes. The size of the buffer used for storing undo information can be configured in the File -> Preferences dialog; by default it allows 7 steps backward at any one time.

**Center Horizontally**  The Center Horizontally operation sets the selected objects’ X position specification to "50%", their alignment to center, and fixes them in that horizontal position. To "unfix" an object, uncheck the "fix" box in its dialog box. Center vertically naturally works in just the same way for the y position.

**Align Horizontally**  This operation sets the X position and alignment of all the selected objects to the X position and alignment of the first selected object. Whether the objects end up appearing to be left-, center-, or right-aligned will depend on the alignment of the first selected object. "Align vertically" works just the same way.

**Grid**  To use the Grid function, place several items roughly in a grid, select them all, perform the operation, and hopefully they will be nicely aligned.

**Copy**  The Copy function simply creates a duplicate of each selected object.

**Equalize widths**  This function simply copies the width of the first selected object to all of the other selections. "Equalize heights" naturally does the same for heights.

**Even Space Horizontally**  This operation moves the objects so that they are equally spaced between the leftmost and rightmost objects of the current selections, which are not moved. "Even Space Vertically" does the same vertically.

**Reorder**  The Reorder function is used to reorder the selected objects in the program’s internal lists. This is useful so that they are produced in the program output in the desired order, for example to ensure that the tab key moves from object to object in the right sequence. By selecting several objects and using reorder, those objects appear first in sequence in the order in which they were selected.
18.13 Summary

The Unicon GUI toolkit offers a full-featured, attractive way of constructing interfaces. The toolkit has many features—tables, tabbed property sheets, and multi-line text editing capability—that are not present in Icon’s vidgets GUI library. Many components present in both libraries are more flexible in the GUI toolkit, supporting fonts, colors, and graphics that the vidgets library does not handle. The ivib interface builder tool provides programmers with easy access to the GUI toolkit.

Object-orientation mainly affects this class library’s extensibility, although it also contributes to the simplicity of the design. Inheritance, including multiple inheritance, is used extensively in the 37 classes of the GUI toolkit. Inheritance is the main object-oriented feature that could not be easily mimicked in a procedural toolkit such as the vidgets library, and inheritance is the primary extension mechanism.
Part IV
Appendices
Appendix A

Language Reference

Unicon is expression-based. Nearly everything is an expression, including the common control structures such as while loops. The only things that are not expressions are declarations for procedures, methods, variables, records, classes, and linked libraries.

In the reference, types are listed for parameters and results. If an identifier is used, any type is allowed. For results, generator expressions are further annotated with an asterisk (*) and non-generators that can fail are annotated with a question mark (?). A question mark by itself (short for null?) denotes a predicate whose success or failure is what matters; the predicate return value ($\&null$) is not significant.

A.1 Immutable Types: Numbers, Strings, Csets

Unicon’s immutable types are integers, real numbers, strings, and csets. Values of these types cannot change. Operators and functions on immutable types produce new values rather than modify existing ones. The simplest expressions are literal values, which occur only for immutable types. A literal value evaluates to itself.

Integer

Integers are of arbitrary precision. Decimal integer literals are contiguous sequences of the digits 0 through 9, optionally preceded by a + or - sign. Suffixes K, M, G, T, or P multiply a literal by 1024, 1024^2, 1024^3, 1024^4, and 1024^5, respectively.

Radix integer literals use the format radixRdigits, where radix is a base in the range 2 through 36, and digits consists of one or more numerals in the supplied radix. After values 0-9, the letters A-Z are used for values 10-35. Radix literals are case insensitive, unlike the rest of the language, so the R may be upper or lower case, as may the following alphabetic digits.
Real

Reals are double-precision floating-point values. Real decimal literals are contiguous sequences of the digits 0 through 9, with a decimal point (a period) somewhere within or at either end of the digits. Real exponent literals use the format numberEinteger; E may be upper or lower case.

String

Strings are sequences of 0 or more characters, where a character is a value with a platform-dependent size and symbolic representation. On platforms with multi-byte character sets, multiple Icon characters represent a single symbol using a platform-dependent encoding. String literals consist of 0 or more characters enclosed in double quotes. A string literal may include escape sequences that use multiple characters to encode special characters. The escape sequences are given in Table A-1. Incomplete string literals may be continued on the next line if the last character on a line is an underscore (_). In that case, the underscore, the newline, and any whitespace at the beginning of the next line are not part of the string literal.

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>backspace</td>
<td>\d</td>
<td>delete</td>
<td>\e</td>
<td>escape</td>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>\l</td>
<td>line feed</td>
<td>\n</td>
<td>newline</td>
<td>\r</td>
<td>carriage return</td>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
<td>'</td>
<td>quote</td>
<td>&quot;</td>
<td>double quote</td>
<td>\</td>
<td>backslash</td>
</tr>
<tr>
<td>\ooo</td>
<td>octal</td>
<td>\xhh</td>
<td>hexadecimal</td>
<td>^x</td>
<td>Control-x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cset

Csets are sets of 0 or more characters. Cset literals consist of 0 or more characters enclosed in single quotes. As with strings, a cset literal may include escape sequences that use multiple characters to encode special characters.

A.2 Mutable Types: Containers and Files

Mutable types’ values may be altered. Changes to a mutable value affect its allocated memory or its associated OS resource. Mutable types include lists, tables, sets, records, objects, and files, including windows, network connections and databases. These types are described in the entries for constructors that create them. Structure types hold collections of elements that may be of arbitrary, mixed type.
A.3. VARIABLES

List

Lists are dynamically sized, ordered sequences of zero or more values. They are constructed by function, by an explicit operator, or implicitly by a call to a variable argument procedure. They change size by stack and queue functions.

Table

Tables are dynamically sized, unordered mappings from keys to elements. They are constructed by function. The keys may be of arbitrary, mixed type.

Set

Sets are unordered collections. They are constructed by function.

Record

Records are ordered, fixed length sequences of elements accessed via named fields.

Object

Objects are ordered, fixed length sequences of elements that may be accessed via named fields and methods. Accessing an object’s fields from outside its methods (using it as a record) is legal but deprecated.

File

Files are system resources corresponding to data on secondary storage, areas on users’ displays, network connections, or databases. Operations on files cause input or output side effects on the system outside of the program execution.

A.3 Variables

Variables are names for locations in memory where values can be stored. Values are stored in variables by assignment operators. A variable name begins with a letter or underscore, followed by zero or more letters, underscores, or digits. A variable name cannot be the same as one of Icon’s reserved words, nor can it be the same as one of Icon’s keywords if it follows an adjacent ampersand character. Variables can hold values of any type, and may hold different types of values at different times during program execution.

There are four kinds of variables: global, local, static, and class. Global, local, and static variables are declared by introducing one of the reserved words (global, local, or
static) followed by a comma-separated list of variable names. Global variables are declared outside of any procedure or method body, while local and static variables are declared at the beginning of procedure and method bodies. Local and static variable names may be followed by an assignment operator and an initial value; otherwise variables other than procedure and class names begin with the value $null$.

Aliasing occurs when two or more variables refer to the same value, such that operations on one variable might affect the other. Aliasing is a common source of program bugs. Variables holding integer, real, string, or cset values are never aliased, because those types are immutable.

Global

Global variables are visible everywhere in the program, and exist at the same location for the entire program execution. Declaring a procedure declares a global variable initialized to the procedure value that corresponds to the code for that procedure.

Local

Local variables exist and are visible within a single procedure or method only for the duration of a single procedure invocation, including suspensions and resumptions, until the procedure returns, fails, or is vanquished by the return or failure of an ancestor invocation while it is suspended. Undeclared variables in any scope are implicitly local, but this dangerous practice should be avoided in large programs.

Variables that are declared as parameters are local variables that are preinitialized to the values of actual parameters at the time of a procedure or method invocation. The semantics of parameter passing are the same as those of assignment.

Static

Static variables are visible only within a single procedure or method, but exist at the same location for the entire program execution. The value stored in a static variable is preserved between multiple calls to the procedure in which it is declared.

Class

Class variables are visible within the methods of a declared class. Class variables are created for each instance (object) of the class. The lifespan of class variables is the life span of the instance to which they belong. The value stored in a class variable is preserved between multiple calls to the methods of the class in which it is declared.
### A.4 Keywords

Keywords are names with global scope and special semantics within the language. They begin with an ampersand character. Some keywords are names of common constant values, while others are names of variables that play a special role in Icon's control structures. The name of the keyword is followed by a : if it is read-only, or a := if it is a variable, followed by the type of value the keyword holds.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;allocated : integer*</td>
<td>report memory use</td>
<td>&amp;allocated generates the cumulative number of bytes allocated in heap, static, string, and block regions during the entire program execution.</td>
</tr>
<tr>
<td>&amp;ascii : cset</td>
<td>ASCII character set</td>
<td>&amp;ascii produces a cset corresponding to the ASCII characters.</td>
</tr>
<tr>
<td>&amp;clock : string</td>
<td>time of day</td>
<td>&amp;clock produces a string consisting of the current time of day in hh:mm:ss format. See also keyword &amp;now.</td>
</tr>
<tr>
<td>&amp;collections : integer*</td>
<td>garbage collection activity</td>
<td>&amp;collections generates the number of times memory has been reclaimed in heap, static, string, and block regions.</td>
</tr>
<tr>
<td>&amp;column : integer</td>
<td>source code column</td>
<td>&amp;column returns the source code column number of the current execution point. This is especially useful for execution monitoring.</td>
</tr>
<tr>
<td>&amp;cset : cset</td>
<td>universal character set</td>
<td>&amp;cset produces a cset constant corresponding to the universal set of all characters.</td>
</tr>
<tr>
<td>&amp;current :co-expression</td>
<td>current co-expression</td>
<td>&amp;current produces the co-expression that is currently executing.</td>
</tr>
<tr>
<td>&amp;date : string</td>
<td>today’s date</td>
<td>&amp;date produces the current date in yyyy/mm/dd format.</td>
</tr>
<tr>
<td>&amp;dateline : string</td>
<td>time stamp</td>
<td>&amp;dateline produces a human-readable time stamp that includes the day of the week, the date, and the current time, down to the minute.</td>
</tr>
<tr>
<td>&amp;digits : cset</td>
<td>digit characters</td>
<td>&amp;digits produces a cset constant corresponding to the set of digit characters 0-9.</td>
</tr>
<tr>
<td>&amp;dump := integer</td>
<td>termination dump</td>
<td></td>
</tr>
</tbody>
</table>
&dump controls whether the program dumps information on program termination or not. If &dump is nonzero when the program halts, a dump of local and global variables and their values is produced.

\[
\&e : \text{real} \\
\&e \text{ is the base of the natural logarithms, } 2.7182818...
\]

\[
\&error := \text{integer} \\
\&error \text{ controls whether runtime errors are converted into expression failure. By assigning to this keyword, error conversion can be enabled or disabled for specific sections of code. The integer } \&error \text{ is decremented by one on each error, and if it reaches zero, a runtime error is generated. Assigning a value of -1 effectively disables runtime errors indefinitely. See also } \&\text{syserr}.
\]

\[
\&errornumber : \text{integer?} \\
\&errornumber \text{ is the error number of the last runtime error that was converted to failure, if there was one.}
\]

\[
\&errortext : \text{string?} \\
\&errortext \text{ is the error message of the last error that was converted to failure.}
\]

\[
\&errorvalue : \text{any?} \\
\&errorvalue \text{ is the erroneous value of the last error that was converted to failure.}
\]

\[
\&errout : \text{file} \\
\&errout \text{ is the standard error file. It is the default destination to which runtime errors and program termination messages are written.}
\]

\[
\&eventcode := \text{integer} \\
\&eventcode \text{ indicates the kind of behavior that occurred in a monitored program at the time of the most recent call to } \text{EvGet}(). \text{ This keyword is only supported under interpreters built with execution monitoring support.}
\]

\[
\&eventsourcet : \text{co-expression} \\
\&eventsourcet \text{ is the co-expression that transmitted the most recent event to the current program. This keyword is null unless the program is an execution monitor. See also } \&\text{source}. \text{ Under a monitor coordinator, } \&\text{eventsourcet} \text{ is the coordinator and global variable Monitored is the target program.}
\]

\[
\&eventvalue := \text{any} \\
\&eventvalue \text{ is a value from the monitored program that was being processed at the time of the last program event returned by } \text{EvGet}(). \text{ This keyword is only supported under interpreters built with execution monitoring support.}
\]

\[
\&fail : \text{none} \\
\&fail \text{ is an expression failure}
\]
&fail never produces a result. Evaluating it always fails.

<table>
<thead>
<tr>
<th><strong>&amp;features : string</strong></th>
<th>platform features</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;features generates strings that indicate the non-portable features supported on the current platform.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;file : string</strong></th>
<th>current source file</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;file is the name of the source file for the current execution point, if there is one. This is especially useful for execution monitoring.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;host : string</strong></th>
<th>host machine name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;host is a string that identifies the host computer Icon is running on.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;input : file</strong></th>
<th>standard input file</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;input is a standard input file. It is the default source for file input functions.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;lcase : cset</strong></th>
<th>lowercase letters</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lcase is a cset consisting of the lowercase letters from a to z.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;letters : cset</strong></th>
<th>letters</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;letters is a cset consisting of the upper and lowercase letters A-Z and a-z.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;level : integer</strong></th>
<th>call depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;level gives the nesting level of the currently active procedure call. This keyword is not supported under the optimizing compiler, iconc.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;line : integer</strong></th>
<th>current source line number</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;line is the line number in the source code that is currently executing.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;main : co-expression</strong></th>
<th>main task</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;main is the co-expression in which program execution began.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;now : integer</strong></th>
<th>current time</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;now produces the current time as the number of seconds since the epoch beginning 00:00:00 GMT, January 1, 1970. See also &amp;clock</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;null : null</strong></th>
<th>null value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;null produces the null value.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;output : file</strong></th>
<th>standard output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;output is the standard output file. It is the default destination for file output.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;phi : real</strong></th>
<th>golden ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;phi is the golden ratio, 1.618033988...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&amp;pi : real</strong></th>
<th>pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;pi</td>
<td></td>
</tr>
</tbody>
</table>

A.4. **KEYWORDS**
&pi is the value of pi, 3.141592653...

---

**&pos := integer**

*string scanning position*

&pos is the position within the current subject of string scanning. It is assigned implicitly by entering a string scanning environment, moving or tabbing within the environment, or assigning a new value to &subject. &pos may not be assigned a value that is outside the range of legal indices for the current &subject string.

---

**&proname := string**

*program name*

&proname is the name of the current executing program.

---

**&random := integer**

*random number seed*

&random is the seed for random numbers produced by the random operator, unary ? . It is assigned a different sequence for each execution but may be explicitly set for reproducible results.

---

**&regions : integer***

*region sizes*

&regions produces the sizes of the static region, the string region, and the block region. The first result is zero; it is included for backward compatibility reasons.

---

**&source : co-expression**

*invoking co-expression*

&source is the co-expression that activated the current co-expression.

---

**&storage : integer***

*memory in use*

&storage gives the amount of memory currently used within the static region, the string region, and the block region. The first result is always zero and is included for backward compatibility reasons.

---

**&subject := string**

*string scanning subject*

&subject holds the default value used in string scanning and analysis functions. Assigning to &subject implicitly assigns the value 1 to &pos.

---

**&syserr := integer**

*halt on system error*

&syserr controls whether a system error causes execution to halt. System errors cause expression failure by default. If &syserr is set to a non-zero value, system errors are converted to runtime errors and halt execution with an error traceback.

---

**&time : integer**

*elapsed time*

&time gives the number of milliseconds of CPU time that have elapsed since the program execution began. For wall clock time see &now or &clock.

---

**&trace := integer**

*trace program*

&trace gives the number of nesting levels to which program execution will be traced. 0 means no tracing. A negative value traces to an infinite depth. &trace is set outside the program using the TRACE environment variable or the -t compiler option.
A.4. KEYWORDS

&ucase : cset
&ucase is a cset consisting of all the upper case letters from A to Z.

&version : string
&version is a string that indicates which version of Unicon or Icon is executing.

Graphics keywords

Several of the graphics keywords are variables with assignment restricted to value of a particular type or types. Graphics keywords are more fully described in [Griswold98].

&col : integer
&col is the mouse location in text columns during the most recent Event(). If &col is assigned, &x gets a corresponding pixel location in the current font on &window.

&control : integer
&control produces the null value if the control key was pressed at the time of the most recently processed event, otherwise &control fails.

&interval : integer
&interval produces the time between the most recently processed event and the event that preceded it, in milliseconds.

&ldrag : integer
&ldrag produces the integer that indicates a left button drag event.

&lpress : integer
&lpress produces the integer that indicates a left button press event.

&lrelease : integer
&lrelease produces the integer that indicates a left button release event.

&mdrag : integer
&mdrag produces the integer that indicates a middle button drag event.

&meta : integer
&meta produces the null value if the meta (Alt) key was pressed at the time of the most recently processed event, otherwise &meta fails.

&mpress : integer
&mpress produces the integer that indicates a middle button press event.

&mrelease : integer
&mrelease produces the integer that indicates a middle button release event.
APPENDIX A. LANGUAGE REFERENCE

&pick : string*
&pick generates the object IDs selected at point (&x,&y) at the most recent Event(), if the event was read from a 3D window with the attribute pick=on.

&rdrag : integer
&rdrag produces the integer that indicates a right button drag event.

&resize : integer
&resize produces the integer that indicates a window resize event.

&rrow : integer
&rrow is the mouse location in text rows during the most recent Event(). If &rrow is assigned, &y gets a corresponding pixel location in the current font on &window.

&rpress : integer
&rpress produces the integer that indicates a right button press event.

&rrelease : integer
&rrelease produces the integer that indicates a right button release event.

&shift : integer
&shift produces the null value if the shift key was pressed at the time of the most recently processed event, otherwise &shift fails.

&window : window
&window is the default window argument for all window functions. &window may be assigned any value of type window.

&x : integer
&x is the horizontal mouse location in pixels during the most recent Event(). If &x is assigned, &col gets a corresponding text coordinate in the current font on &window.

&y : integer
&y is the vertical mouse location in pixels during the most recent Event(). If &y is assigned, &row gets a corresponding text coordinate in the current font on &window.

A.5 Control Structures and Reserved Words

Unicon has many reserved words. Some are used in declarations, but most are used in control structures. This section summarizes the syntax and semantics introduced by all the reserved words of the language. The reserved word under discussion is written in a bold font. The surrounding syntax uses square brackets for optional items and an asterisk for items that may repeat.
A.5. CONTROL STRUCTURES AND RESERVED WORDS

**break expr**

The `break` expression exits the nearest enclosing loop. `expr` is evaluated and treated as the result of the entire loop expression. If `expr` is another `break` expression, multiple loops will be exited.

**expr1 to expr2 by expr3**

The `by` reserved word supplies a step increment to a `to`-expression (the default is 1).

**case expr of { ? }**

The `case` expression selects one of several branches of code to be executed.

**class name [:: superclass]* (fields) methods [initially] end**

The `class` declaration introduces a new object type into the program. The `class` declaration may include lists of superclasses, fields, methods, and an initially section.

**create expr**

The `create` expression produces a new co-expression to evaluate `expr`.

**critical x : expr**

The `critical` expression serializes the execution of `expr` on value `x`. Value `x` must be a mutex or protected object that has a mutex. The critical section causes `x` to be locked before evaluating `expr` and unlocked afterward. Breaking, returning or failing out of `expr` does not automatically unlock `x`.

**default : expr**

The `default` branch of a case expression is taken if no other case branch is taken.

**do expr**

The `do` reserved word specifies an expression to be executed for each iteration of a preceding `while`, `every`, or `suspend` loop (yes, `suspend` is a looping construct).

**if expr1 then expr2 else expr3**

The `else` expression is executed if `expr1` fails to produce a result.

**end**

The reserved word `end` signifies the end of a procedure, method, or class body.

**every expr1 [do expr2]**

The `every` expression always fails, causing `expr1` to be resumed for all its results.

**fail**

The `fail` reserved word causes the enclosing procedure or method invocation to terminate immediately and produce no results. The invocation may not be resumed. See also the
keyword `&fail`, which produces a less drastic expression failure. `fail` is equivalent to `return &fail`

<table>
<thead>
<tr>
<th><strong>global var [, var]</strong>*</th>
<th>declare global variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved word <code>global</code> introduces one or more global variables.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>if expr then expr2 [else expr3]</strong></th>
<th>conditional expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>if</code> expression evaluates <code>expr2</code> only if <code>expr1</code> produces a result.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>import name [ , name]</strong>*</th>
<th>import package</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>import</code> declaration introduces the names from package <code>name</code> so that they may be used without prefixing them with the package name.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>initial expr</strong></th>
<th>execute on first invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>initial</code> expression is executed the first time a procedure or method is invoked.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>initially [(parameters)]</strong></th>
<th>initialize object</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>initially</code> section defines a special method that is invoked automatically when an object is created. If the <code>initially</code> section has declared parameters, they are used as the parameters of the constructor for objects of that class.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>invocable procedure [ , procedure]</strong>*</th>
<th>allow string invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>invocable all</td>
<td>allow string invocation</td>
</tr>
<tr>
<td>The <code>invocable</code> declaration indicates that procedures may be used in string invocation.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>link filename [ , filename]</strong>*</th>
<th>link code module</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>link</code> declaration directs that the code in <code>filename</code> will be added to the executable when this program is linked. <code>filename</code> may be an identifier or a string literal file path.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>local var :=initializer [ , var := initializer ] [ ]</strong>*</th>
<th>declare local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>local</code> declaration introduces local variables into the current procedure or method. Variable declarations must be at the beginning of a procedure or method.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>method name (params) body end</strong></th>
<th>declare method</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>method</code> declaration introduces a procedure that is invoked with respect to instances of a given class. The <code>params</code> and <code>body</code> are as in procedures, described below.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>next</strong></th>
<th>iterate loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>next</code> expression causes a loop to immediate skip to its next iteration.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>not expr</strong></th>
<th>negate expression failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>not</code> expression fails if <code>expr</code> succeeds, and succeeds (producing null) if <code>expr</code> fails.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>case expr of { ? }</strong></th>
<th>introduce case branches</th>
</tr>
</thead>
</table>
The of reserved word precedes a special compound expression consisting of a sequence of case branches of the form expr : expr. Case branches are evaluated in sequence until one matches the expression given between the word case and the of.

```plaintext
package name declare package
```
The package declaration segregates the global names in the current source file. In order to refer to them, client code must either import the package, or prepend name. (the package name followed by a period) onto the front of a name in the package.

```plaintext
procedure name (params) body end declare procedure
```
The procedure declaration specifies a procedure with parameters and code body. The parameters are a comma-separated list of zero or more variable names. The last parameter may be suffixed by [ ] to indicate that following parameters will be supplied to the procedure in a list. The body is an optional sequence of local and static variable declarations, followed by a sequence of zero or more expressions.

```plaintext
record name (fields) declare record
```
The record declaration introduces a new record type into the program.

```plaintext
repeat expr infinite loop
```
The repeat expression introduces an infinite loop that will reevaluate expr forever. Of course, expr may exit the loop or terminate the program in any number of ways.

```plaintext
return expr return from invocation
```
The return expression exits a procedure or method invocation, producing expr as its result. The invocation may not be resumed.

```plaintext
static var [, var]* declare static variables
```
The static declaration introduces local variables that persist for the entire program execution into the current procedure or method body. Variable declarations must be at the beginning of a procedure or method.

```plaintext
suspend expr [do expr] produce result from invocation
```
The suspend expression produces one or more results from an invocation for use by the calling expression. The procedure or method may be resumed for additional results if the calling expression needs them. Execution in the suspended invocation resumes where it left off, in the suspend expression. A single evaluation of a suspend expression may produce multiple results for the caller if expr is a generator. An optional do expression is evaluated each time the suspend is resumed.

```plaintext
if expr1 then expr2 conditional expression
```
The expr2 following a then is evaluated only if expr1 following an if succeeds. In that case, the result of the whole expression is the result of expr2.
The `thread expr` expression creates and launches a concurrent thread to evaluate `expr`.

The `to` expression produces the integer sequence from `expr1` to `expr2`.

The `until expr1 [do expr2]` expression loops as long as `expr1` fails.

The `while expr1 [do expr2]` expression loops as long as `expr1` succeeds.

### A.6 Operators and Built-in Functions

Icon’s built-ins operators and functions utilize automatic type conversion to provide flexibility and ease of programming. Automatic type conversions are limited to integer, real, string, and cset data types. Conversions to a "number" will convert to either an integer or a real, depending whether the value to be converted has a decimal. Conversions between numeric types and csets go through an intermediate conversion to a string value and are not generally useful.

Indexes start at 1. Index 0 is the position after the last element of a string or list. Negative indexes are positions relative to the end. Subscripting operators and string analysis functions can take two indices to specify a section of the string or list. When two indices are supplied, they select the same string section whether they are in ascending or descending order.

**Operators**

The result types of operators are the same as the operand types except as noted.

**Unary operators**

The `! x : any*` operator produces the elements of `x`. If `x` is a string variable or refers to a structure value, the generated elements are variables that may be assigned. If `i` is equivalent to `(1 to i)` for integer `i`. List, record, string, and file elements are generated in order, with string elements consisting of one-letter substrings. Set and table elements are generated in an undefined order. If `x` is a messaging connection to a POP server, `!x` produces complete messages as strings. Other types of files, including network connections, produce elements consisting of text lines.
A.6. OPERATORS AND BUILT-IN FUNCTIONS

The null and nonnull tests succeed and produce their operand if it satisfies the test.

- number
+ number
Negation reverses the sign of its operand. Numeric identity does not change its operand’s value other than to convert to a required numeric type.

= string
The tab/match operator is equivalent to calling \texttt{tab(match(s))} on its operand.

* \( x \): integer
The size operator returns the number of elements in string, cset or structure \( x \).

. \( x \): \( x \)
The dereference operator returns the value \( x \).

? \( x \): any
The random operator produces a random element from \( x \). If \( x \) is a string, \(?x\) produces a random one-letter substring. The result is a variable that may be assigned. If \( x \) is a positive integer, \(?x\) produces a random integer between 1 and \( x \). \(?0\) returns a real in the range from 0.0-1.0.

\( \mid x \): \( x^* \)
The repeated alternation operator generates results from evaluating its operand over and over again in an infinite loop.

\( \sim \) cset
The complement operator returns a cset consisting of all characters not in its operand.

\( \sim \) co-expression
The refresh operator restarts a co-expression so the next time it is activated it will begin with its first result.

Binary operators

Most binary operators may be augmented with an assignment. If such an operator is followed by a := the left operand must be a variable, and the expression \( x \ op := y \) is equivalent to \( x := x \ op y \). For example, \( x :+: 5 \) is equivalent but faster than the expression \( x := x+5 \).
The arithmetic operators may be augmented.

\[
\begin{align*}
\text{number1} / \text{number2} & \quad \text{divide} \\
\text{number1} \% \text{number2} & \quad \text{modulo} \\
\text{number1} + \text{number2} & \quad \text{add} \\
\text{number1} - \text{number2} & \quad \text{subtract}
\end{align*}
\]

The set operators work on sets or csets. They may be augmented.

\[
\begin{align*}
\text{set1} ** \text{set2} & \quad \text{intersection} \\
\text{set1} ++ \text{set2} & \quad \text{union} \\
\text{set1} -- \text{set2} & \quad \text{difference}
\end{align*}
\]

The field operator selects field name out of a record, object, or package. For objects, name may be a method, in which case the field operator is being used as part of a method invocation. Superclass method invocation consists of a dollar sign and superclass name prior to the field operator.

\[
\begin{align*}
\text{x} . \text{name} & \quad \text{field} \\
\text{object} . \text{name} (\text{params}) & \quad \text{method invocation} \\
\text{object} \$ \text{superclass} . \text{name} (\text{params}) & \quad \text{superclass method invocation}
\end{align*}
\]

Relational operators produce their right operand if they succeed. They may be augmented.

\[
\begin{align*}
\text{number1} = \text{number2} & \quad \text{equal} \\
\text{number1} \sim= \text{number2} & \quad \text{not equal} \\
\text{number1} < \text{number2} & \quad \text{less than} \\
\text{number1} \leq \text{number2} & \quad \text{less or equal} \\
\text{number1} > \text{number2} & \quad \text{greater than} \\
\text{number1} \geq \text{number2} & \quad \text{greater or equal} \\
\text{string1} == \text{string2} & \quad \text{string equal} \\
\text{string1} \sim== \text{string2} & \quad \text{string not equal} \\
\text{string1} <<= \text{string2} & \quad \text{string less than} \\
\text{string1} <== \text{string2} & \quad \text{string less or equal} \\
\text{string1} >>= \text{string2} & \quad \text{string greater than} \\
\text{string1} >>= \text{string2} & \quad \text{string greater or equal} \\
\text{x1} === \text{x2} & \quad \text{equivalence} \\
\text{x1} \sim=== \text{x2} & \quad \text{non equivalence}
\end{align*}
\]

The several assignment operators all require variables for their left operands, and swap operators also require variables for their right operands.
### A.6. OPERATORS AND BUILT-IN FUNCTIONS

#### String scanning

String scanning evaluates `expr` with `&subject` equal to `string` and `&pos` starting at 1. It may be augmented.

#### Binary bang (exclamation) operator

The binary bang (exclamation) operator calls `x`, using `y` as its parameters. `x` may be a procedure, or the string name of a procedure. `y` is a list or record.

#### Activate co-expression

The activate operator transfers execution control from the current co-expression to its right operand co-expression. The transmitted value is `x`, or `&null` if no left operand is supplied. Activation may be augmented.

#### Send and blocking send message

The send operator places a message in another thread’s public inbox, or in the current thread’s public outbox. The normal version fails if the box is full; the blocking version waits for space to become available.

#### Receive and blocking receive message

The receive operator obtains a message from another thread’s public outbox, or the current thread’s public inbox. The normal version fails if the box is empty; the blocking version waits for a message to become available.

#### Concatenation

The concatenation operators produce new values containing a copy of the left operand followed by a copy of the right operand. They may be augmented.

#### Conjunction and alternation

The conjunction operator produces `x2` if `x1` succeeds. The alternation operator produces the results of `expr1` followed by the results of `expr2`; it is a generator. These operators may be augmented.

#### Limitation

The limitation operator fails if it is resumed after its left operand has produced a number of results equal to its right operand.

#### Mutual evaluation

The mutual evaluation operator executes `expr` and `expr2` in parallel.

#### Invocation

The invocation operator executes `expr` in parallel with the rest of the program.
By themselves, parentheses are used to override operator precedence in surrounding expressions. A comma-separated list of expressions is evaluated left to right, and fails if any operand fails. Its value is the right of the rightmost operand.

When preceded by an operand, parentheses form an invocation. The operand may be a procedure, a method, a string that is converted to a procedure name, or an integer that selects the parameter to use as the result of the entire expression.

With no preceding operand, square brackets create and initialize lists. Initializer values are comma-separated, except in list comprehension where the expression’s values (obtained as if by every) are used to provide the initial list elements. When preceded by an operand, square brackets form a subscript or subsection. Multiple comma-separated subscript operands are equivalent to separate subscript operations with repeating square brackets, so \( x[y,z] \) is equivalent to \( x[y][z] \).

Subscripting selects an element from a structure and allows that element to be assigned or for its value to be used. Lists and strings are subscripted using 1-based integer indices, tables are subscripted using arbitrary keys, and records may be subscripted by either string fieldname or 1-based integer index. Message connections may be subscripted by string header to obtain server responses; POP connections may also be subscripted by 1-based integer message numbers.

Subsectioning works on strings and lists. For strings, the subsection is a variable if the string was a variable, and assignment to the subsection makes the variable hold the new, modified string constructed by replacing the subsection. For lists, a subsection is a new list that contains a copy of the elements from the original list.

\[ \text{empty list creation} \]
\[ [ \text{expr}, \text{expr}]* ] \text{list creation} \]
\[ [: \text{expr}:] \text{list comprehension} \]
\[ \text{expr1} [ \text{expr2}, \text{expr}]* \text{subscript} \]
\[ \text{expr1} [ \text{expr2} : \text{expr3} ] \text{subsection} \]
\[ \text{expr1} [ \text{expr2} :+: \text{expr3} ] \text{forward relative subsection} \]
\[ \text{expr1} [ \text{expr2} -: \text{expr3} ] \text{backward relative subsection} \]

With no preceding operand, square brackets create and initialize lists. Initializer values are comma-separated, except in list comprehension where the expression’s values (obtained as if by every) are used to provide the initial list elements. When preceded by an operand, square brackets form a subscript or subsection. Multiple comma-separated subscript operands are equivalent to separate subscript operations with repeating square brackets, so \( x[y,z] \) is equivalent to \( x[y][z] \).

Subscripting selects an element from a structure and allows that element to be assigned or for its value to be used. Lists and strings are subscripted using 1-based integer indices, tables are subscripted using arbitrary keys, and records may be subscripted by either string fieldname or 1-based integer index. Message connections may be subscripted by string header to obtain server responses; POP connections may also be subscripted by 1-based integer message numbers.

Subsectioning works on strings and lists. For strings, the subsection is a variable if the string was a variable, and assignment to the subsection makes the variable hold the new, modified string constructed by replacing the subsection. For lists, a subsection is a new list that contains a copy of the elements from the original list.

\[ \text{expr1} ; \text{expr2} \text{bound expression} \]

A semicolon bounds \( \text{expr1} \). Once \( \text{expr2} \) is entered, \( \text{expr1} \) cannot be resumed for more results. The result of \( \text{expr2} \) is the result of the entire expression. Semicolons are automatically inserted at ends of lines wherever it is syntactically allowable to do so. This results in many implicitly bounded expressions.

\[ \{ \text{expr} ; \text{expr}* \} \text{compound expression} \]
\[ p \{ \text{expr} ; \text{expr}* \} \text{programmer defined control structure} \]

Curly brackets typically cause a sequence of bounded expressions to be treated as a single expression. Preceded by a procedure value, curly brackets introduce a programmer defined
control structure in which a co-expression is created for each argument; the procedure is
called with these co-expressions as its parameters, and can determine for itself whether,
and in what order, to activate its parameters to obtain values.

Built-in functions

Unicon’s built-in functions are a key element of its ease of learning and use. They provide
substantial functionality in a consistent and easily memorized manner.

In addition to automatic type conversion, built-in functions make extensive use of op-
tional parameters with default values. Default values are indicated in the function descrip-
tions, with the exception of string scanning functions. String scanning functions end with
three parameters that default to the string &subject, the integer &pos, and the end of
string (0) respectively. The position argument defaults to 1 when the string argument is
supplied rather than defaulted.

### abs(N) : number

**absolute value**

`abs(N)` produces the maximum of `N` or `-N`.

### acos(r) : real

**arc cosine**

`acos(r)` produces the arc cosine of `r`. The argument is given in radians.

### any(c, s, i, i) : integer?

**cset membership**

String scanning function `any(c,s,i1,i2)` produces `i1+1` if `s[i1:i2][1]` is in cset `c`, but fails
otherwise.

### args(x,i) : any

**number of arguments**

`args(p)` produces the number of arguments expected by procedure `p`. If `p` takes a variable
number of arguments, `args(p)` returns a negative number to indicate that the final argument
is a list conversion of an arbitrary number of arguments. For example, `args(p)` for a
procedure `p` with formal parameters `(x, y, z[])` returns a `-3`. `args(C)` produces the number
of arguments in the current operation in co-expression `C`, and `args(C,i)` produces argument
number `i` within co-expression `C`.

### asin(real) : real

**arc sine**

`asin(r1)` produces the arc sine of `r1`. The argument is given in radians.

### atan(r, r:1.0) : real

**arc tangent**

`atan(r1)` produces the arc tangent of `r1`. `atan(r1,r2)` produces the arc tangent of `r1` and `r2`.
Arguments are given in radians.

### atanh(r) : real

**inverse hyperbolic tangent**

`atanh(r)` produces the inverse hyperbolic tangent of `r`. Arguments are given in radians.
bal(cs:&cset, cs:'(', cs:')', s, i, i) : integer*  
balance string
String scanning function \texttt{bal(c1,c2,c3,s,i1,i2)} generates the integer positions in \texttt{s} at which a member of \texttt{c1} in \texttt{s[i1:i2]} is balanced with respect to characters in \texttt{c2} and \texttt{c3}.

\begin{verbatim}
center(s, i:1, s:" ") : string  
center string
center(s1,i,s2) produces a string of \texttt{i} characters. If \texttt{i > *s1} then \texttt{s1} is padded equally on the left and right with \texttt{s2} to length \texttt{i}. If \texttt{i < *s1} then the center \texttt{i} characters of \texttt{s1} are produced.
\end{verbatim}

\begin{verbatim}
channel(TH) : list  
communications channel
channel(TH) creates a communications channel between the current thread and thread \texttt{TH}.
\end{verbatim}

\begin{verbatim}
char(i) : string  
encode character
char(i) produces a string consisting of the character encoded by integer \texttt{i}.
\end{verbatim}

\begin{verbatim}
chdir(s) : string  
change directory
chdir(s) changes the current working directory to \texttt{s}. \texttt{chdir()} returns the current working directory.
\end{verbatim}

\begin{verbatim}
chmod(f, m) : ?  
file permissions
\texttt{chmod(f, m)} sets the access permissions ("mode") of a string filename (or on UNIX systems, an open file) \texttt{f} to a string or integer mode \texttt{m}. The mode indicates the change to be performed. The string is of the form

\texttt{[ugoa][+-=][rwxRWXstugo]*}

The first group describes the set of mode bits to be changed: \texttt{u} is the owner set, \texttt{g} is the group and \texttt{o} is the set of all others. The character \texttt{a} designates all the fields. The operator \texttt{(+-=)} describes the operation to be performed: \texttt{+} adds a permission, \texttt{-} removes a permission, and \texttt{=} sets a permission. The permissions themselves are:

\begin{verbatim}
r       read
w       write
x       execute
R       read if any other set already has r
W       write if any other set already has w
X       execute if any other set already has x
s       setuid (if the first part contains u and/or setgid if the first part contains g
t       sticky if the first part has o
u       the u bits on the same file
g       the g bits on the same file
o       the o bits on the same file
\end{verbatim}

If the first group is missing, then it is treated as "all" except that any bits in the user’s umask will not be modified in the mode. Not all platforms make use of all mode bits
A.6. OPERATORS AND BUILT-IN FUNCTIONS

Described here; the mode bits that are used is a property of the filesystem on which the file resides.

classname(r) : string
classname(r) produces the name of r’s class.

| close(f) : file | integer | close file |
close(f) closes file, pipe, window, network or message connection, or database f and returns any resources associated with it to the operating system. If f was a window, close(f) causes it to disappear, but the window can still be written to and copied from until all open bindings are closed. If f was a pipe or network connection, close() returns the integer exit status of the connection, otherwise it returns the closed file.

| cofail(CE) : any | transmit co-expression failure |
cofail(ce) activates co-expression ce, transmitting failure instead of a result.

| collect(i:0, i:0) : null | collect garbage |
collect(i1,i2) calls the garbage collector to ensure that i2 bytes are free in region i1. i1 can be 0 (no region in particular) 1 (static region) 2 (string region) or 3 (block region).

| condvar() : condition variable | create condition variable |
condvar() creates a new condition variable.

| constructor(s, ...) : procedure | record constructor |
constructor(label, field, field, ...) creates a new record type named label with fields named by its subsequent arguments, and returns a constructor procedure for this record type.

| copy(any) : any | copy value |
copy(x) produces a copy of x. For immutable types (numbers, strings, csets, procedures) this is a no-op. For mutable types (lists, tables, sets, records, objects) a one-level deep copy of the object is made.

| cos(r) : real | cosine |
cos(r) produces the cosine of r. The argument is given in radians.

| cset(any) : cset? | convert to cset |
cset(x) converts x to a cset, or fails if the conversion cannot be performed.

| ctime(i) : string | format a time value into local time |
ctime(i) converts an integer time given in seconds since the epoch, Jan 1, 1970 00:00:00 into a string in the local timezone. See also keywords &clock and &dateline.

| dbcolumns(D,s) : list | ODBC column information |
dbcolums(db, tablename) produces a list of record (catalog, schema, tablename, colname, datatype, typename, colsizze, buflen, decdigits, numprecradix, nullable, remarks) entries.
Fields **datatype** and **typename** are SQL-dependent and data source dependent, respectively. Field **colsize** gives the maximum length in characters for SQL_CHAR or SQL_VARCHAR columns. Field **decdigits** gives the number of significant digits right of the decimal. Field **numprecradix** specifies whether **colsize** and **decdigits** are specified in bits or decimal digits.

Field **nullable** is 0 if the column does not accept null values, 1 if it does accept null values, and 2 if it is not known whether the column accepts null values.

---

**dbdriver(D) : record**

**dbdriver(db)** produces a record (driver(name, ver, odbcver, connections, statements, dsn)) that describes the details of the ODBC driver used to connect to database db. Connections and statements are the maximums the driver can support. Fields **ver** and **odbcver** are the driver and ODBC version numbers. Fields **name** and **dsn** are the driver filename and Windows Data Source Name associated with the connection.

---

**dbkeys(D,string) : list**

**dbkeys(db,tablename)** produces a list of record (columnname, sequencenumber) pairs containing information about the primary keys in tablename.

---

**dblimits(D) : record**


---

**dbproduct(D) : record**

**dbproduct(db)** produces a record (name, ver) that gives the name and the version of the DBMS product containing db.

---

**dbtables(D) : list**

**dbtables(db)** returns a list of record (qualifier, owner, name, type, remarks) entries that describe all of the tables in the database associated with db.

---

**delay(i) : null**

**delay(i)** pauses the program for at least i milliseconds.

---

**delete(x1, x2, ...) : x1**

**delete(x1, x2)** deletes elements denoted by the 2nd and following parameters from set, table, list, DBM database, or POP connection x1 if it is there. In any case, it returns x1. If x1 is a table or set, elements x_i denote keys of arbitrary type. If x1 is a DBM database, indices must be strings. If x1 is a list or a POP messaging connection, elements x_i are integer indices of the element to be deleted. POP messages are actually deleted when the close() operation closes that connection.
A.6. OPERATORS AND BUILT-IN FUNCTIONS

\texttt{detab(string, integer:9,...)} : string \hspace{1cm} \text{replace tabs}

\texttt{detab(s,i,...)} replaces tabs with spaces, with stops at columns indicated by the second and following parameters, which must all be integers. Tab stops are extended infinitely using the interval between the last two specified tab stops.

\texttt{display(i:&level, f:&errout, CE:&current)} : null \hspace{1cm} \text{write variables}

\texttt{display(i,f)} writes the local variables of \texttt{i} most recent procedure activations, plus global variables, to file \texttt{f}.

\texttt{dtor(r)} : real \hspace{1cm} \text{convert degrees to radians}

\texttt{dtor(r)} produces the equivalent of \texttt{r} degrees, expressed in radians.

\texttt{entab(s, i:9,...)} : string \hspace{1cm} \text{replace spaces}

\texttt{entab(s,i,...)} replaces spaces with tabs, with stops at columns indicated. Tab stops are extended infinitely using the interval between the last two specified tab stops.

\texttt{errorclear()} : null \hspace{1cm} \text{clear error condition}

\texttt{errorclear()} resets keywords \&errornumber, \&errortext, and \&errorvalue to indicate that no error is present.

\texttt{eventmask(CE, cset)} : cset | null \hspace{1cm} \text{get/set event mask}

\texttt{eventmask(ce)} returns the event mask associated with the program that created \texttt{ce}, or \&null if there is no event mask. \texttt{eventmask(ce,cs)} sets that program’s event mask to \texttt{cs}.

\texttt{EvGet(c, flag)} : string \hspace{1cm} \text{get event from monitored program}

\texttt{EvGet(c,flag)} activates a program being monitored until an event in \texttt{cset} mask \texttt{c} occurs. Under normal circumstances this is a one-character string event code.

\texttt{EvSend(i, x, CE)} : any \hspace{1cm} \text{transmit event}

\texttt{EvSend(x, y, C)} transmits an event with event code \texttt{x} and event value \texttt{y} to a monitoring co-expression \texttt{C}.

\texttt{exit(i:normalexit)} \hspace{1cm} \text{exit process}

\texttt{exit(i)} terminates the current program execution, returning status code \texttt{i}. The default is the platform-dependent exit code that indicates normal termination (0 on most systems).

\texttt{exp(r)} : real \hspace{1cm} \text{exponential}

\texttt{exp(r)} produces the result of \&e \^ r.

\texttt{fetch(D, s?)} : string | row? \hspace{1cm} \text{fetch database value}

\texttt{fetch(d, k)} fetches the value corresponding to key \texttt{k} from a DBM or SQL database \texttt{d}. The result is a string (for DBM databases) or a row (for SQL databases). For SQL databases, when the string \texttt{k} is omitted, \texttt{fetch(d)} produces the next row in the current selection, and
advances the cursor to the next row. A row is a record whose field names and types are determined by the columns specified in the current query. `fetch(d)` fails if there are no more rows to return from the current query. Typically a call to `dbselect()` will be followed by a while-loop that calls `fetch()` repeatedly until it fails.

---

**fieldnames(R) : string***

Get field names

`fieldnames(r)` produces the names of the fields in record `r`.

---

**find(s, s, i, i) : integer***

Find string

String scanning function `find(s1,s2,i1,i2)` generates the positions in `s2` at which `s1` occurs as a substring in `s2[i1:i2]`.

---

**flock(f, s) : ?**

Apply or remove file lock

`flock(f,s)` applies an advisory lock to the file. Advisory locks enable processes to cooperate when accessing a shared file, but do not enforce exclusive access. The following characters can be used to make up the operation string:

- `s` shared lock
- `x` exclusive lock
- `b` don’t block when locking
- `u` unlock

Locks cannot be applied to windows, directories or database files. A file may not simultaneously have shared and exclusive locks.

---

**flush(f) : file**

Flush file

`flush(f)` flushes all pending or buffered output to file `f`.

---

**function() : string***

Name the functions

`function()` generates the names of the built-in functions.

---

**get(L,i:1) : any?**

Get element from queue

`get(L)` returns an element which is removed from the head of the queue `L`. `get(L, i)` removes `i` elements, returning the last one removed.

---

**getch() : string?**

Get character from console

`getch()` waits for (if necessary) and returns a character typed at the keyboard, even if standard input was redirected. The character is not displayed.

---

**getche() : string?**

Get and echo character from console

`getche()` waits for (if necessary) and returns a character typed at the console keyboard, even if standard input was redirected. The character is echoed to the screen.

---

**getenv(s) : string?**

Get environment variable

`getenv(s)` returns the value of environment variable `s` from the operating system.
gettimeofday() : record
time of day
Returns the current time in seconds and microseconds since the epoch, Jan 1, 1970 00:00:00. The sec value may be converted to a date string with ctime or gtime. See also keywords &now, &clock, and &dateline. Return value: record posix_timeval(sec, usec)

globalnames(CE) : string*
name the global variables
globalnames(ce) generates the names of the global variables in the program that created co-expression ce.

gtime(i) : string
format a time value into UTC
Converts an integer time in seconds since the epoch, Jan 1, 1970 00:00:00 into a string in Coordinated Universal Time (UTC).

iand(i, i) : integer
bitwise and
iand(i1, i2) produces the bitwise AND of i1 and i2.

icom(i) : integer
bitwise complement
icom(i) produces the bitwise complement (one’s complement) of i.

image(any) : string
string image
image(x) returns the string image of the value x.

insert(x1, x2, x3:&null) : x1
insert element
insert(x1, x2, x3) inserts element x2 into set, table, or list or DBM database x1 if not already there. Unless x1 is a set, the assigned value for element x2 is x3. For lists, x2 is an integer index; for other types, it is a key. insert() always succeeds and returns x1.

integer(any) : integer?
convert to integer
integer(x) converts value x to an integer, or fails if the conversion cannot be performed.

ior(i, i) : integer
bitwise or
ior(i1, i2) produces the bitwise OR of i1 and i2.

ishift(i, i) : integer
bitwise shift
ishift(i, j) produces the value obtained by shifting i by j bit positions. Shifting is to the left if j>0, or to the right if j<0. j zero bits are introduced at the end opposite the shift direction.

istate(CE, s) : integer
interpreter state
istate(ce, attrib) reports selected virtual machine interpreter state information. attrib must be one of: “count”, “ilevel”, “ipc”, “ipc_offset”, “sp”, “efp”, “gfp”. Used by monitors.

ixor(i, i) : integer
bitwise xor
ixor(i1, i2) produces the bitwise exclusive or of i1 and i2.
APPENDIX A. LANGUAGE REFERENCE

kbhit() : ?
check for console input
kbhit() checks to see if there is a keyboard character waiting to be read.

key(x) : any*
table keys
key(T) generates the key (entry) values from table T. key(L) generates the indices from 1 to *L in list L. key(R) generates the string field names of record R.

keyword(s,CE:&current,i:0) : any*
produce keyword value
keyword(s,ce,i) produces the value of keyword s in the context of ce’s execution, i levels up in the stack from the current point of execution. Used in execution monitors.

left(s, i:1, s:” ”) : string
left format string
left(s1,i,s2) formats s1 to be a string of length i. If s1 is more than i characters, it is truncated. If s1 is fewer than i characters it is padded on the right with as many copies of s2 as needed to increase it to length i.

list(integer:0, any:&null) : list
create list
list(i, x) creates a list of size i, in which all elements have the initial value x. If x is a mutable value such as a list, all elements refer to the same value, not a separate copy of the value for each element.

load(s,L,f:&input,f:&output,f:&errout,i,i,i) : co-expression
load Unicon program
load(s,arglist,input,output,error,blocksize,stringsize,stacksize) loads the icode file named s and returns that program’s execution as a co-expression ready to start its main() procedure with parameter arglist as its command line arguments. The three file parameters are used as that program’s &input, &output, and &errout. The three integers are used as its initial memory region sizes.

loadfunc(s, s) : procedure
load C function
loadfunc(filename,funcname) dynamically loads a compiled C function from the object library file given by filename. funcname must be a specially written interface function that handles Icon data representations and calling conventions.

localnames(CE, i:0) : string*
local variable names
localnames(ce,i) generates the names of local variables in co-expression ce, i levels up from the current procedure invocation. The default i of 0 generates names in the currently active procedure in ce.

lock(x) : x
lock mutex
lock(x) locks the mutex x or the mutex associated with thread-safe object x.

log(r, r:&e) : real
logarithm
A.6. OPERATORS AND BUILT-IN FUNCTIONS

\[ \log(r_1, r_2) \] produces the logarithm of \( r_1 \) to base \( r_2 \).

\[ \text{many}(c, s, i, i) : \text{integer}? \]
String scanning function \( \text{many}(c, s, i_1, i_2) \) produces the position in \( s \) after the longest initial sequence of members of \( c \) within \( s[i_1:i_2] \).

\[ \text{map}(s, s:\&ucase, s:\&lcase) : \text{string} \]
\[ \text{map}(s_1, s_2, s_3) \] maps \( s_1 \), using \( s_2 \) and \( s_3 \). The resulting string will be a copy of \( s_1 \), with the exception that any of \( s_1 \)'s characters that appear in \( s_2 \) are replaced by characters at the same position in \( s_3 \).

\[ \text{match}(s, s:\&subject, i:\&pos, i:0) : \text{integer} \]
String scanning function \( \text{match}(s_1, s_2, i_1, i_2) \) produces \( i_1 + *s_1 \) if \( s_1 == s_2[i_1:*s_1] \), but fails otherwise.

\[ \text{max}(n, ...) : \text{number} \]
\[ \text{max}(x, ...) \] returns the largest value among its arguments, which must be numeric.

\[ \text{member}(x, ...) : x? \]
\[ \text{member}(x, ...) \] returns \( x \) if its second and subsequent arguments are all members of set, cset, list or table \( x \) but fails otherwise. If \( x \) is a cset, all of the characters in subsequent string arguments must be present in \( x \) in order to succeed.

\[ \text{membernames}(x) : \text{list} \]
\[ \text{membernames}(x) \] produces a list containing the string names of the fields of \( x \), where \( x \) is either an object or a string name of a class.

\[ \text{methodnames}(x) : \text{list} \]
\[ \text{methodnames}(x) \] produces a list containing the string names of the methods defined in class \( x \), where \( x \) is either an object or a string name of a class.

\[ \text{methods}(x) : \text{list} \]
\[ \text{methods}(x) \] produces a list containing the procedure values of the methods of \( x \), where \( x \) is either an object or a string name of a class.

\[ \text{min}(n, ...) : \text{number} \]
\[ \text{min}(x, ...) \] returns the smallest value among its arguments, which must be numeric.

\[ \text{mkdir}(s, s?) : ? \]
\[ \text{mkdir}(\text{path}, \text{mode}) \] creates a new directory named \( \text{path} \) with mode \( \text{mode} \). The optional \( \text{mode} \) parameter can be numeric or a string of the form accepted by \( \text{chmod}() \). The function succeeds if a new directory is created.

\[ \text{move}(i:1) : \text{string} \]
\[ \text{move}(i:1) \] produces the scanning position.
move(i) moves &pos i characters from the current position and returns the substring of &subject between the old and new positions. This function reverses its effects by resetting the position to its old value if it is resumed.

mutex(x,y) : x

mutex() creates a new mutex. For mutex(x) associates the new mutex with structure x. The call mutex(x,y) associates an existing mutex y (or mutex associated with protected object y) with structure x.

ame(v, CE:&current) : string

name(v) returns the name of variable v within the program that created co-expression c. Keyword variables are recognized and named correctly. name() returns the base type and subscript or field information for variables that are elements within other values, but does not produce the source code variable name for such variables.

numeric(any) : number

numeric(x) produces an integer or real number resulting from the type conversion of x, but fails if the conversion is not possible.

open(s, s:”rt”, ...) : file?

open(s1, s2, ...) opens a file named s1 with mode s2 and attributes given in trailing arguments. The modes recognized by open() are:

"a" append; write after current contents
"b" open for both reading and writing (b does not mean binary mode!)
"c" create a new file and open it
"d" open a [NG]DBM database
"g" create a 2D graphics window
"gl" create a 3D graphics window
"n" connect to a remote TCP network socket
"na" accept a connection from a TCP network socket
"nau" accept a connection from a UDP network socket
"nl" listen on a TCP network socket
"nu" connect to a UDP network socket
"m" connect to a messaging server (HTTP, SMTP, POP, ...)
"o" open an ODBC connection to a (typically SQL) database
"p" execute a program given by command line s1 and open a pipe to it
"r" read
"t" use text mode, with newlines translated
"u" use a binary untranslated mode
"w" write

Directories may only be opened for reading, and produce the names of all files, one per line. Pipes may be opened for reading or writing, but not both.
When opening a network socket: the first argument \texttt{s1} is the name of the socket to connect. If \texttt{s1} is of the form "s:i", it is an Internet domain socket on host \texttt{s} and port \texttt{i}; otherwise, it is the name of a Unix domain socket. If the host name is null, it represents the current host. Mode "n" allows an optional third parameter, an integer timeout (in milliseconds) after which \texttt{open()} fails if no connection has been established by that time.

For a UDP socket, there is not really a connection, but any writes to that file will send a datagram to that address, so that the address doesn’t have to be specified each time. Also, \texttt{read()} or \texttt{reads()} cannot be performed on a UDP socket; use receive. UDP sockets must be in the INET domain; the address must have a colon.

For a DBM database, only one modifier character may be used: if \texttt{s1} is "dr" it indicates that the database should be opened in read-only mode. For an ODBC database, following the mode letter "o" comes an optional string default table name used by functions such as \texttt{dbcolumns()}, followed by two generally required strings giving the username and password authentication for the connection.

The filename argument is a Uniform Resource Indicator (URI) when opening a messaging connection. Arguments after the mode "m" are sent as headers. The HTTP User-Agent header defaults to "Unicon Messaging/10.0" and Host defaults to the host and port indicated in the URI. The SMTP From: header obtains its default from a UNICON_USERADDRESS environment variable if it is present.

For 2D and 3D windows, attribute values may be specified in the following arguments to \texttt{open()}. \texttt{open()} fails if a window cannot be opened or an attribute cannot be set to a requested value.

\begin{verbatim}
opmask(CE, c) : cset                       opcode mask
  opmask(ce) gets ce's program's opcode mask. The function returns &null if there is no opcode mask. opmask(ce,cs) sets ce's program's opcode mask to cs. This function is part of the execution monitoring facilities.

oprec(x) : record                        get methods vector
  operc(r) produces a variable reference for r's class' methods vector.

ord(s) : integer                       ordinal value
  ord(s) produces the integer ordinal (value) of s, which must be of size 1.

paramnames(CE, i:0) : string*          parameter names
  paramnames(ce,i) produces the names of the parameters in the procedure activation i levels above the current activation in ce.

parent(CE) : co-expression            parent program
  parent(ce) returns &main for ce's parent program. This is interesting only when programs are dynamically loaded using the load() function.
\end{verbatim}
pipe() : list  
create pipe

pipe() creates a pipe and returns a list of two file objects. The first is for reading, the second is for writing. See also function filepair().

pop(L | Message) : any?  
pop from stack

pop(L) removes an element from the top of the stack (L[1]) and returns it. pop(M) removes and returns the first message in POP mailbox connection M; the actual deletion occurs when the messaging connection is closed.

pos(i) : integer?  
test scanning position

pos(i) tests whether &pos is at position i in &subject.

proc(any, i:1, C) : procedure?  
convert to procedure

proc(s,i) converts s to a procedure if that is possible. Parameter i is used to resolve ambiguous string names; it must be either 0, 1, 2, or 3. If i is 0, a built-in function is returned if it is available, even if the global identifier by that name has been assigned differently. If i is 1, 2, or 3, the procedure for an operator with that number of operands is produced. For example, proc("-",2) produces the procedure for subtraction, while proc("-") produces the procedure for unary negation. proc(C,i) returns the procedure activated i levels up with C. proc(p, i, C) returns procedure p if it belongs to the program which created co-expression C.

pull(L,i:1) : any?  
remove from list end

pull(L) removes and produces an element from the end of a nonempty list L. pull(L, i) removes i elements, producing the last one removed.

push(L, any, ...) : list  
push on to stack

push(L, x1, ..., xN) pushes elements onto the beginning of list L. The order of the elements added to the list is the reverse of the order they are supplied as parameters to the call to push(). push() returns the list that is passed as its first parameter, with the new elements added.

put(L, x1, ..., xN) : list  
add to list end

put(L, x1, ..., xN) puts elements onto the end of list L.

read(f:&input) : string?  
read line

read(f) reads a line from file f. The end of line marker is discarded.

reads(f:&input, i:1) : string?  
read characters

reads(f,i) reads up to i characters from file f. It fails on end of file. If f is a network connection, reads() returns as soon as it has input available, even if fewer than i characters were delivered. If i is -1, reads() reads and produces the entire file as a string. Care should be exercised when using this feature to read very large files.
A.6. OPERATORS AND BUILT-IN FUNCTIONS

ready(f:&input, i:0) : string?  non-blocking read
ready(f,i) reads up to i characters from file f. It returns immediately with available data and fails if no data is available. If i is 0, ready() returns all available input. It is not currently implemented for window values.

real(any) : real?  convert to real
real(x) converts x to a real, or fails if the conversion cannot be performed.

receive(f) : record  receive datagram
receive(f) reads a datagram addressed to the port associated with f, waiting if necessary. The returned value is a record of type posix_message(addr, msg), containing the address of the sender and the contents of the message respectively.

remove(s) : ?  remove file
remove(s) removes the file named s.

rename(s, s) : ?  rename file
rename(s1,s2) renames the file named s1 to have the name s2.

repl(x, i) : x  replicate
repl(x, i) concatenates and returns i copies of string or list x.

reverse(x) : x  reverse sequence
reverse(x) returns a value that is the reverse of string or list x.

right(s, i:1, s:"") : string  right format string
right(s1,i,s2) produces a string of length i. If i<s1, s1 is truncated. Otherwise, the function pads s1 on left with s2 to length i.

rmdir(s) : ?  remove directory
rmdir(d) removes the directory named d. rmdir() fails if d is not empty or does not exist.

rtod(r) : real  convert radians to degrees
rtod(r) produces the equivalent of r radians, expressed in degrees.

runerr(i, any)  runtime error
runerr(i,x) produces runtime error i with value x. Program execution is terminated.

seek(f, any) : file?  seek to file offset
seek(f,i) seeks to offset i in file f, if it is possible. If f is a regular file, i must be an integer. If f is a database, i seeks a position within the current set of selected rows. The position is selected numerically if i is convertible to an integer; otherwise i must be convertible to a string and the position is selected associatively by the primary key.
select(x1, x2, ?) : list
files with available input
select(files?, timeout) waits for an input to become available on any of several files, typically
network connections or windows. Its arguments may be files or lists of files, ending with
an optional integer timeout value in milliseconds. It returns a list of those files among its
arguments that have input waiting.

If the final argument to select() is an integer, it is an upper bound on the time elapsed
before select returns. A timeout of 0 causes select() to return immediately with a list of
files on which input is currently pending. If no files are given, select() waits for its timeout
to expire. If no timeout is given, select() waits forever for available input on one of its file
arguments. Directories and databases cannot be arguments to select().

send(s, s) : ?
send datagram
send(s1, s2) sends a UDP datagram to the address s1 (in host:port format) with the
contents s2.

seq(i:1, i:1) : integer*
generate sequence
seq(i, j) generates the infinite sequence i, i+j, i+2*j, ... . j may not be 0.

serial(x) : integer?
structure serial number
serial(x) returns the serial number for structure x, if it has one. Serial numbers uniquely
identify structure values.

set(x, ...) : set
create set
set() creates a set. Arguments are inserted into the new set, with the exception of lists.
set(L) creates a set whose members are the elements of list L.

setenv(s) : ?
set environment variable
setenv() sets an environment variable s in the operating system.

signal(cv, i:1) : ??
signal a conditional variable
signal(x, y) signals the condition variable x. If y is supplied, the condition variable is
signaled y times. If y is 0, a “broadcast” signal is sent waking up all threads waiting on x.

sin(r) : real
sine
sin(r) produces the sine of r. The argument is given in radians.

sort(x, i:1) : list
sort structure
sort(x, i) sorts structure x. If x is a table, parameter i is the sort method. If i is 1 or 2, the
table is sorted into a list of lists of the form [key, value]. If i is 3 or 4, the table is sorted
into a list of alternating keys and values. Sorting is by keys for odd-values of i, and by
table element values for even-values of i.

sortf(x, i:1) : list
sort by field
**A.6. OPERATORS AND BUILT-IN FUNCTIONS**

sortf(x,i) sorts a list, record, or set x using field i of each element that has one. Elements that don't have an i’th field are sorted in standard order and come before those that do have an i’th field.

spawn(CE, i, i) : thread
spawn(ce) launches co-expression ce as an asynchronous thread that will execute concurrently with the current co-expression. The two optional integers specify the memory in bytes allocated for the thread’s block and string regions. The defaults are 10% of the main thread heap size.

sql(D, s) : integer
sql(db, query) executes arbitrary SQL code on db. This function allows the program to do vendor-specific SQL and many SQL statements that cannot be expressed otherwise using the Unicon database facilities. sql() can leave the database in an arbitrary state and should be used with care.

sqrt(r) : real
sqrt(r) produces the square root of r.

stat(f) : record?
stat(f) returns a record with information about the file f which may be a path or a file object. The return value is of type: record posix_stat(dev, ino, mode, nlink, uid, gid, rdev, size, atime, mtime, ctime, blksize, blocks, symlink). Many of these fields are POSIX specific, but a number are supported across platforms, such as the file size in bytes (the size field), access permissions (the mode field), and the last modified time (the mtime field).

The atime, mtime, and ctime fields are integers that may be formatted with the ctime() and gtime() functions. The mode is a string similar to the long listing option of the UNIX ls(1) command. For example, "-rwrxrwsr-x" represents a plain file with a mode of 2775 (octal). stat(f) fails if filename or path f does not exist.

staticnames(CE:&current, i:0) : string*
staticnames(ce,i) generates the names of static variables in the procedure i levels above the current activation in ce.

stop(s|f, ...) :
stop(args) halts execution after writing out its string arguments, followed by a newline, to &errout. If any argument is a file, subsequent string arguments are written to that file instead of &errout. The program exit status indicates that an error has occurred.

string(x) : string?
string(x) converts x to a string and returns the result, or fails if the value cannot be converted.
system(x, f:&input, f:&output, f:&errout, s) : integer execute system command
system(x, f1, f2, f3, waitflag) launches execution of a program in a separate process. x can be either a string or a list of strings. In the former case, whitespace is used to separate the arguments and the command is processed by the platform’s command interpreter. In the second case, each member of the list is an argument and the second and subsequent list elements are passed unmodified to the program named in the first element of the list.

The three file arguments are files that will be used for the new process’ standard input, standard output and standard error. The return value is the exit status from the process. If the waitflag argument is "nowait", system() returns immediately after spawning the new process, and the return value is then the process id of the new process.

sys_errstr(i) : string? system error string
sys_errstr(i) produces the error string corresponding to i, a value obtained from &errno.

tab(i:0) : string? set scanning position
tab(i) sets &pos to i and returns the substring of &subject spanned by the former and new positions. tab(0) moves the position to the end of the string. This function reverses its effects by resetting the position to its old value if it is resumed.

table(k,v, ..., x) : table create table
table(x) creates a table with default value x. If x is a mutable value such as a list, all references to the default value refer to the same value, not a separate copy for each key. Given more than one argument, table(k,v,...x) takes alternating keys and values and populates the table with these initial contents.

tan(r) : real tangent
tan(r) produces the tangent of r in radians.

trap(s, p) : procedure trap or untrap signal
trap(s, proc) sets up a signal handler for the signal s (the name of the signal). The old handler (if any) is returned. If proc is null, the signal is reset to its default value.

Caveat: This is not supported by the optimizing compiler (the -C command line option, which invokes iconc).

trim(s, c:' ', i:-1) : string trim string
trim(s,c,i) removes characters in c from s at the back (i=-1, the default), at the front (i=1), or at both ends (i=0).

truncate(f, i) : ? truncate file
truncate(f, len) changes the file f (which may be a string filename, or an open file) to be no longer than length len. truncate() does not work on windows, network connections, pipes, or databases.
trylock(x) : x?
trylock(x) locks the mutex x or the mutex associated with thread-safe object x, if it is not locked.

type(x) : string
type(x) returns a string that indicates the type of x.

unlock(x) : x
unlock(x) unlocks the mutex x or the mutex associated with thread-safe object x.

upto(c, s, i, i) : integer*
String scanning function upto(c,s,i1,i2) generates the sequence of integer positions in s up to a character in c in s[i2:i2], but fails if there is no such position.

utime(s, i, i) : null
utime(f, atime, mtime) sets the access time for a file named f to atime and the modification time to mtime. The ctime is set to the current time. The effects of this function are platform specific. Some file systems support only a subset of these times.

variable(s, CE:&current, i:0) : any?
variable(s, c, i) finds the variable with name s and returns a variable descriptor that refers to its value. The name s is searched for within co-expression c, starting with local variables i levels above the current procedure frame, and then among the global variables in the program that created c.

wait(x) : ?
wait(x) waits for x. If x is a thread, wait() waits for it to finish. If x is a condition variable wait() waits until that variable is signaled by another thread.

where(f) : integer?
where(f) returns the current offset position in file f. It fails on windows and networks. The beginning of the file is offset 1.

write(s|f, ...) : string|file
write(args) outputs strings, followed by a newline, to a file or files. Strings are written in order to their nearest preceding file, defaulting to &output. A newline is output to the preceding file after the last argument, as well as whenever a non-initial file argument directs output to a different file. write() returns its last argument.

writes(s|f, ...) : string|file
writes(args) outputs strings to one or more files. Each string argument is written to the nearest preceding file argument, defaulting to &output. writes() returns its last argument.
Graphics functions

The names of built-in graphics functions begin with upper case. The built-in graphics functions are listed here. These functions are more thoroughly described in [Griswold98]. Extensive procedure and class libraries for graphics are described in [Griswold98] and in Appendix B. In 2D, arguments named \(x\) and \(y\) are pixel locations in zero-based integer coordinates. In 3D windows coordinates are given using real numbers, and functions by default take three coordinates \((x,y,z)\) per vertex. Attribute \texttt{dim} can be set to 2 or 4, changing most 3D functions to take vertices in a \((x,y)\) or \((x,y,z,w)\) format. Arguments named \texttt{row} and \texttt{col} are cursor locations in one-based integer text coordinates. Most functions' first parameter named \texttt{w} defaults to \&\texttt{window} and the window argument can be omitted in the default case.

\begin{tabular}{ll}
\hline
\textbf{Active() : window} & produce active window \\
\textbf{Active()} & returns a window that has one or more events pending. If no window has an event pending, \textbf{Active()} blocks and waits for an event to occur. \textbf{Active()} starts with a different window on each call in order to avoid window "starvation". \textbf{Active()} fails if no windows are open.
\end{tabular}

\begin{tabular}{ll}
\hline
\textbf{Alert() : window} & alert the user \\
\textbf{Alert()} & produces a visual flash or audible beep that signifies to the user the occurrence of some notable event in the application.
\end{tabular}

\begin{tabular}{ll}
\hline
\textbf{Bg(w,s) : string} & background color \\
\textbf{Bg(w)} & retrieves the background color. \textbf{Bg(w,s)} sets the background color by name, rgb, or mutable color value. \textbf{Bg()} fails if the background cannot be set to the requested color.
\end{tabular}

\begin{tabular}{ll}
\hline
\textbf{Clip(w,x:0,y:0,width:0,height:0) : window} & clip to rectangle \\
\textbf{Clip(w,x,y,width,height)} & clips output to a rectangular area within the window. If \texttt{width} is 0, the clip region extends from \texttt{x} to the right side of the window. If \texttt{height} is 0, the clip region extends from \texttt{y} to the bottom of the window.
\end{tabular}

\begin{tabular}{ll}
\hline
\textbf{Clone(w,s,...) : window} & clone context \\
\textbf{Clone(w)} & produces a new window binding in which a new graphics context is copied from \texttt{w} and bound to \texttt{w}'s canvas. Additional string arguments specify attributes of the new binding, as in \texttt{WAttrib()}. If the first string argument is "g" or "gl", \textbf{Clone()} binds the new context to a subwindow with separate canvas and input queue inside of and relative to \texttt{w}. \textbf{Clone()} fails if an attribute cannot be set to a requested value.
\end{tabular}

\begin{tabular}{ll}
\hline
\textbf{Color(w, i, s,...) : window} & set mutable color \\
\textbf{Color(w,i)} & produces the current setting of mutable color \texttt{i}. \textbf{Color(w,i,s,...)} sets the color map entries identified by \texttt{i[j]} to the corresponding colors \texttt{s[j]}. See [Griswold98].
\end{tabular}
A.6. OPERATORS AND BUILT-IN FUNCTIONS

ColorValue(w, s): string  
convert color name to rgb

ColorValue(w, s) converts the string color s into a string with three comma-separated 16-bit integer values denoting the color’s RGB components. ColorValue() fails if string s is not a valid name or recognized decimal or hex encoding of a color.

CopyArea(w1, w2, x:0, y:0, width:0, height:0, x2:0, y2:0): window  
copy area

CopyArea(w1, w2, x, y, width, height, x2, y2) copies a rectangular region within w1 defined by x,y,width,height to window w2 at offset x2,y2. CopyArea() returns w1. &window is not a default for this function. The default copies all of w1.

Couple(w1, w2): window  
couple window to context

Couple(w1, w2) produces a new value that binds the window associated with w1 to the graphics context associated with w2.

DrawArc(w, x, y, width, height, a1:0.0, a2:2*π, ...): window  
draw arc

DrawArc(w, x, y, width, height, a1, a2, ...) draws arcs or ellipses. Each is defined by six integer coordinates. x, y, width and height define a bounding rectangle around the arc; the center of the arc is the point \((x+(\text{width})/2, y+(\text{height})/2)\). Angles are specified in radians. Angle a1 is the starting position of the arc, where 0.0 is the 3 o’clock position and the positive direction is counter-clockwise. Angle a2 is not the end position, but rather specifies the direction and extent of the arc.

DrawCircle(w, x, y, radius, a1:0.0, a2:2*π, ...): window  
draw circle

DrawCircle() draws a circle or arc, centered at \((x,y)\) and otherwise similar to DrawArc() with \text{width=height}.

DrawCube(w, x, y, z, len ...): record  
draw cube

DrawCube(w, x, y, z, len...) draws a cube with sides of length len at the position \((x, y, z)\) on the 3D window w. The display list element is returned. This procedure fails if the context attribute \text{dim} is set to 2.

DrawCurve(w, x1, y1, ...): window  
draw curve

DrawCurve(w, x1, y1, ..., xn, yn) draws a smooth curve connecting each \(x,y\) pair in the argument list. If the first and last point are the same, the curve is smooth and closed through that point.

DrawCylinder(w, x, y, z, h, r1, r2, ...): record  
draw cylinder

DrawCylinder(w, x, y, z, h, r1, r2, ...) draws a cylinder with a top of radius r1, a bottom with radius r2, and a height h on 3D window w. The disk is centered at the point \((x, y, z)\). The display list element is returned. This procedure fails if the context attribute \text{dim} is set to 2.
**APPENDIX A. LANGUAGE REFERENCE**

**DrawDisk**(\(w, x, y, z, r_1, r_2, a_1, a_2, \ldots\)) : record
draw disk

**DrawDisk**(\(W, x, y, z, r_1, r_2, a_1, a_2, \ldots\)) draws a disk or partial disk centered at \((x, y, z)\) on 3D window \(w\). The inner circle has radius \(r_1\) and the outer circle has radius \(r_2\). The parameters \(a_1\) and \(a_2\) are optional. If they are specified, a partial disk is drawn with a starting angle \(a_1\) and sweeping angle \(a_2\). The display list element is returned.

**DrawImage**(\(w, x, y, s\)) : window
draw bitmapped figure

**DrawImage**(\(w, x, y, s\)) draws an image specified in string \(s\) at location \(x, y\).

**DrawLine**(\(w, x_1, y_1, \ldots\)) : window [list]
draw line

**DrawLine**(\(w, x_1, y_1, \ldots\)) draws lines between each adjacent \(x, y\) pair of arguments. In 3D, **DrawLine()** takes from 2-4 coordinates per vertex and returns the list that represents the lines on the display list for refresh purposes.

**DrawPoint**(\(w, x_1, y_1, \ldots\)) : window [list]
draw point

**DrawPoint**(\(w, x_1, y_1, \ldots\)) draws points. In 3D, **DrawPoint()** takes from 2-4 coordinates per vertex and returns the list that represents the points on the display list for refresh purposes.

**DrawPolygon**(\(w, x_1, y_1, [z_1,] \ldots\)) : window [list]
draw polygon

**DrawPolygon**(\(w, x_1, y_1, [z_1,] \ldots\)) draws a polygon. In 3D, **DrawPolygon()** takes from 2-4 coordinates per vertex and returns the list that represents the polygon on the display list for refresh purposes.

**DrawRectangle**(\(w, x_1, y_1, width_1, height_1, \ldots\)) : window
draw rectangle

**DrawRectangle**(\(w, x_1, y_1, width_1, height_1, \ldots\)) draws rectangles. Arguments \(width\) and \(height\) define the perceived size of the rectangle; the actual rectangle drawn is \(width+1\) pixels wide and \(height+1\) pixels high.

**DrawSegment**(\(w, x_1, y_1, [z_1,] \ldots\)) : window [list]
draw line segment

**DrawSegment**(\(w, x_1, y_1, [z_1,] \ldots\)) draws lines between alternating \(x, y\) pairs in the argument list. In 3D, **DrawSegment()** takes from 2-4 coordinates per vertex and returns the list that represents the segments on the display list for refresh purposes.

**DrawSphere**(\(w, x, y, z, r, \ldots\)) : record
draw sphere

**DrawSphere**(\(w, x, y, z, r, \ldots\)) draws a sphere with radius \(r\) centered at \((x, y, z)\) on 3D window \(w\). The display list element is returned. This procedure fails if the context attribute \(dim\) is set to 2.

**DrawString**(\(w, x_1, y_1, s_1, \ldots\)) : window
draw text

**DrawString**(\(w, x_1, y_1, s_1, \ldots\)) draws text \(s\) at coordinates \((x, y)\). This function does not draw any background; only the characters' actual pixels are drawn. It is possible to use "drawop=reverse" with this function to draw erasable text. **DrawString()** does not affect the
text cursor position. Newlines present in s cause subsequent characters to be drawn starting at (x, current_y + leading), where x is the x supplied to the function, current_y is the y coordinate the newline would have been drawn on, and leading is the current leading associated with the binding.

\textbf{DrawTorus}(w, x, y, z, r1, r2, ...) : record draw torus
\textbf{DrawTorus}(w, x, y, z, r1, r2, ...) draws a torus with inner radius r1, outside radius r2, and centered at (x,y,z) on 3D window w. The display list element is returned. This procedure fails if the context attribute \texttt{dim} is set to 2.

\textbf{EraseArea}(w, x:0, y:0, width:0, height:0. ...) : window erase rectangular area
\textbf{EraseArea}(w,x,y,width,height,...) erases rectangular areas within the window to the background color. If \texttt{width} is 0, the region cleared extends from x to the right side of the window. If \texttt{height} is 0, the region erased extends from y to the bottom of the window. In 3D, \textbf{EraseArea}(W) clears the contents of the entire window.

\textbf{Event}(w, i:infinity) : string|integer read event on window
\textbf{Event}(w, i) retrieves the next event available for window \texttt{w}. If no events are available, \textbf{Event()} waits for i milliseconds. Keystrokes are encoded as strings, while mouse events are encoded as integers. The retrieval of an event is accompanied by assignments to the keywords \texttt{&x}, \texttt{&y}, \texttt{&row}, \texttt{&col}, \texttt{&interval}, \texttt{&control}, \texttt{&shift}, \texttt{&meta}, and if 3D attribute “pick=on”, \texttt{&pick}. \textbf{Event()} fails if the timeout expires before an event occurs.

\textbf{Fg}(w, s) : string foreground color
\textbf{Fg}(w) retrieves the current foreground color. \textbf{Fg}(w,s) sets the foreground by name or value. \textbf{Fg()} fails if the foreground cannot be set to the requested color. In 3D, \textbf{Fg}(w, s) changes the material properties of subsequently drawn objects to the material properties specified by \texttt{s}. The string \texttt{s} must be one or more semi-colon separated material properties. A material property is of the form

[diffuse | ambient | specular | emission] color name or “shininess n”, 0 <= n <= 128.

If string \texttt{s} is omitted, the current values of the material properties will be returned.

\textbf{FillArc}(w, x, y, width, height, a1, a2, ...) : window draw filled arc
\textbf{FillArc}(w,x,y,width,height,a1,a2,...) draws filled arcs, ellipses, and/or circles. Coordinates are as in \textbf{DrawArc}().

\textbf{FillCircle}(w, x, y, radius, a1, a2, ...) : window draw filled circle
\textbf{FillCircle}(w,x,y,radius,a1,a2,...) draws filled circles. Coordinates are as in \textbf{DrawCircle}().

\textbf{FillPolygon}(w, x1, y1, [z1,] ...) : window draw filled polygon
\textbf{FillPolygon}(w,x1,y1,...,xn,yn) draws a filled polygon. The beginning and ending points are connected if they are not the same. In 3D, \textbf{FillPolygon}() takes from 2-4 coordinates
per vertex and returns the list that represents the polygon on the display list for refresh purposes.

**FillRectangle**(*w, x:0, y:0, width:0, height:0, ...*) : window draw filled rectangle

*FillRectangle*(w,x,y,width,height,...) draws filled rectangles.

**Font**(*w, s*) : string

*Font*(w) produces the name of the current font. *Font*(w,s) sets the window context’s font to s and produces its name or fails if the font name is invalid. The valid font names are largely system-dependent but follow the format family[,styles],size, where styles optionally add bold or italic or both. Four font names are portable: serif (Times or similar), sans (Helvetica or similar), mono (a mono spaced sans serif font) and typewriter (Courier or similar). *Font()* fails if the requested font name does not exist.

**FreeColor**(*w, s, ...*) : window release colors

*FreeColor*(w,s1,...,sn) allows the window system to re-use the corresponding color map entries. Whether this call has an effect is dependent upon the particular implementation. If a freed color is still in use at the time it is freed, unpredictable results will occur.

**GotoRC**(*w, row:1, col:1*) : window go to row,column

*GotoRC*(w,row,col) moves the text cursor to a particular row and column, given in numbers of characters; the upper-left corner is coordinate (1,1). The column calculation used by *GotoRC()* assigns to each column the pixel width of the widest character in the current font. If the current font is of fixed width, this yields the usual interpretation.

**GotoXY**(*w, x:0, y:0*) : window go to pixel

*GotoXY*(w,x,y) moves the text cursor to a specific cursor location in pixels.

**IdentityMatrix**(*w*) : record load the identity matrix

*IdentityMatrix*(w) changes the current matrix to the identity matrix on 3D window w. The display list element is returned.

**Lower**(*w*) : window lower window

*Lower*(w) moves window w to the bottom of the window stack.

**MatrixMode**(*w, s*) : record set matrix mode

*MatrixMode*(w, s) changes the matrix mode to s on 3D window w. The string s must be either “projection” or “modelview”; otherwise this procedure fails. The display list element is returned.

**MultMatrix**(*w, L*) : record multiply transformation matrix

*MultMatrix*(w, L) multiplies the current transformation matrix used in 3D window w by the 4x4 matrix represented as a list of 16 values L.
NewColor(w, s) : integer
allocate mutable color
NewColor(w,s) allocates an entry in the color map and returns a small negative integer for this entry, usable in calls to routines that take a color specification, such as Fg(). If s is specified, the entry is initialized to the given color. NewColor() fails if it cannot allocate an entry.

PaletteChars(w, s) : string
pallele characters
PaletteChars(w,s) produces a string containing each of the letters in palette s. The pallete “c1” through “c6” define different color encodings of images represented as string data; see [Griswold98].

PaletteColor(w, p, s) : string
pallele color
PaletteColor(w,s) returns the color of key s in palette p in “r,g,b” format.

PaletteKey(w, p, s) : integer
pallele key
PaletteKey(w,s) returns the key of closest color to s in palette p.

Pattern(w, s) : w
define stipple pattern
Pattern(w,s) selects stipple pattern s for use during draw and fill operations. s may be either the name of a system-dependent pattern or a literal of the form width,bits. Patterns are only used when the fillstyle attribute is stippled or opaquestippled. Pattern() fails if a named pattern is not defined. An error occurs if Pattern() is given a malformed literal.

Pending(w, x, ...) : L
produce event queue
Pending(w) produces the list of events waiting to be read from window w. If no events are available, the list is empty (its size is 0). Pending(w,x1,...,xn) adds x1 through xn to the end of w’s pending list in guaranteed consecutive order.

Pixel(w, x:0, y:0, width:0, height:0) : i1...in
generate window pixels
Pixel(w,x,y,width,height) generates pixel contents from a rectangular area within window w. width * height results are generated starting from the upper-left corner and advancing down to the bottom of each column before the next one is visited. Pixels are returned in integer values; ordinary colors are encoded nonnegative integers, while mutable colors are negative integers that were previously returned by NewColor(). Ordinary colors are encoded with the most significant eight bits all zero, the next eight bits contain the red component, the next eight bits the green component, and the least significant eight bits contain the blue component. Pixel() fails if part of the requested rectangle extends beyond the canvas.

PopMatrix(w) : record
pop the matrix stack
PopMatrix(w) pops the top matrix from either the projection or modelview matrix stack on 3D window w, depending on the current matrix mode. This procedure fails if there is only one matrix on the matrix stack. The display list element is returned.
APPENDIX A. LANGUAGE REFERENCE

**PushMatrix**($w$)** : record**
push the matrix stack

*PushMatrix*($w$) pushes a copy of the current matrix onto the matrix stack on 3D window $w$. The current matrix mode determines on what stack is pushed. This procedure fails if the stack is full. The "**projection**" stack is of size two; the "**modelview**" stack is of size thirty two. The display list element is returned.

**PushRotate**($w$, $a$, $x$, $y$, $z$) **: record**
push and rotate

*PushRotate*() is equivalent to *PushMatrix()* followed by *Rotate()*.

**PushScale**($w$, $x$, $y$, $z$) **: record**
push and scale

*PushScale*() is equivalent to *PushMatrix()* followed by *Scale()*.

**PushTranslate**($w$, $x$, $y$, $z$) **: record**
push and translate

*PushTranslate*() is equivalent to *PushMatrix()* followed by *Translate()*.

**QueryPointer**($w$) **: x, y**
produce mouse position

*QueryPointer*($w$) generates the $x$ and $y$ coordinates of the mouse relative to window $w$. If $w$ is omitted, *QueryPointer()* generates the coordinates relative to the upper-left corner of the entire screen.

**Raise**($w$) **: window**
raise window

*Raise*($w$) moves window $w$ to the top of the window stack, making it entirely visible and possibly obscuring other windows.

**ReadImage**($w$, $s$, $x$:0, $y$:0) **: integer**
load image file

*ReadImage*($w$, $s$, $x$, $y$) loads an image from the file named by $s$ into window (or texture) $w$ at offset $x$, $y$. $x$ and $y$ are optional and default to 0,0. GIF, JPG, PNG, and BMP formats are supported, along with platform-specific formats. If *ReadImage()* reads the image into $w$, it returns either an integer 0 indicating no errors occurred or a nonzero integer indicating that one or more colors required by the image could not be obtained from the window system. *ReadImage()* fails if file $s$ cannot be opened for reading or is an invalid file format.

**Refresh**($w$) **: window**
redraw the window

*Refresh*($w$) redraws the contents of window $w$. It is used mainly when objects have been moved in a 3D scene. The window $w$ is returned.

**Rotate**($w$, $a$, $x$, $y$, $z$) **: record**
rotate objects

*Rotate*($w$, $a$, $x$, $y$, $z$, $\ldots$) rotates subsequent objects drawn on 3D window $w$ by angle $a$ degrees, in the direction ($x$, $y$, $z$). The display list element is returned.

**Scale**($w$, $x$, $y$, $z$) **: record**
scale objects

*Scale*($w$, $x$, $y$, $z$, $\ldots$) scales subsequent objects drawn on 3D window $w$ according to the given coordinates. The display list element is returned.
### A.6. OPERATORS AND BUILT-IN FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Texcoord(w, x, y, ...) : list</code></td>
<td>define texture coordinates</td>
</tr>
<tr>
<td><code>Texcoord(W, x_1, y_1, ..., x_n, y_n)</code></td>
<td>sets the texture coordinates to $x_1, y_1, ..., x_n, y_n$ in 3D window $w$. Each $x, y$ pair forms one texture coordinate. <code>Texcoord(W, L)</code> sets the texture coordinates to those specified in the list $L$. <code>Texcoord(W, s)</code> sets the texture coordinates to those specified by $s$. The string $s$ must be “auto” otherwise the procedure will fail. In all cases the display list element is returned.</td>
</tr>
<tr>
<td><code>TextWidth(w, s) : integer</code></td>
<td>pixel width of text</td>
</tr>
<tr>
<td><code>TextWidth(w, s)</code></td>
<td>computes the pixel width of string $s$ in the font currently defined for window $w$.</td>
</tr>
<tr>
<td><code>Texture(w, s) : record</code></td>
<td>apply texture</td>
</tr>
<tr>
<td><code>Texture(w, s)</code></td>
<td>specifies a texture image that is applied to subsequent objects drawn on 3D window $w$. The string $s$ specifies the texture image as a filename, a string of the form <code>width,pallet,data</code> or <code>width,#,data</code>, where pallet is a pallet from the Unicon 2D graphics facilities and data is the hexadecimal representation of an image. <code>Texture(w_1, w_2)</code> specifies that the contents of 2D or 3D window $w_2$ be used as a texture image that is applied to subsequent objects on the window $w_1$. The display list element is returned.</td>
</tr>
<tr>
<td><code>Translate(w, x, y, z, ...) : record</code></td>
<td>translate object positions</td>
</tr>
<tr>
<td><code>Translate(w, x, y, z, ...)</code></td>
<td>moves objects drawn subsequently on 3D window $w$ in the direction $(x,y,z)$. The display list element is returned.</td>
</tr>
<tr>
<td><code>Uncouple(w) : window</code></td>
<td>release binding</td>
</tr>
<tr>
<td><code>Uncouple(w)</code></td>
<td>releases the binding associated with file $w$. <code>Uncouple()</code> closes the window only if all other bindings associated with that window are also closed.</td>
</tr>
<tr>
<td><code>WAttrib(w, s1, ...) : x, ...</code></td>
<td>generate or set attributes</td>
</tr>
<tr>
<td><code>WAttrib(w, s1, ...)</code></td>
<td>retrieves and/or sets window and context attributes. If called with exactly one attribute, integers are produced for integer-value attributes; all other values are represented by strings. If called with more than one attribute argument, <code>WAttrib()</code> produces one string result per argument, prefixing each value by the attribute name and an equals sign (=). If $xi$ is a window, subsequent attributes apply to $xi$. <code>WAttrib()</code> fails if an attempt is made to set the attribute <code>font</code>, <code>fg</code>, <code>bg</code>, or <code>pattern</code> to a value that is not supported. A run-time error occurs for an invalid attribute name or invalid value.</td>
</tr>
<tr>
<td><code>WDefault(w, program, option) : string</code></td>
<td>query user preference</td>
</tr>
<tr>
<td><code>WDefault(w, program, option)</code></td>
<td>returns the value of option for program as registered with the X resource manager. In typical use this supplies the program with a default value for window attribute option from a program.option entry in an .XDefaults file. <code>WDefault()</code> fails if no user preference for the specified option is available.</td>
</tr>
</tbody>
</table>
**APPENDIX A. LANGUAGE REFERENCE**

**WFlush(w)** : window  
flush window output

WFlush(w) flushes window output on window systems that buffer text and graphics output. Window output is automatically flushed whenever the program blocks on window input. When this behavior is not adequate, a call to WFlush() sends all window output immediately, but does not wait for all commands to be received and acted upon. WFlush() is a no-op on window systems that do not buffer output.

**WindowContents(w)** : list  
window display list

WindowContents(w) returns a list of display elements, which are records or lists. Each element has a function name followed by the parameters of the function, or an attribute followed by its value.

**WriteImage(w, s, x:0, y:0, width:0, height:0)** : window  
save image file

WriteImage(w, s, x, y, width, height) saves an image of dimensions width, height from window w at offset x, y to a file named s. The default is to write the entire window. The file is written according to the filename extension, in either GIF, JPG, BMP, PNG, or a platform specific format such as XBM or XPM. WriteImage() fails if s cannot be opened for writing.

**WSection(w, s)** : record  
define window section

WSection(w, s) starts a new window section named s on 3D window w and returns a display list section marker record. During window refreshes if the section marker’s skip field is 1, the section is skipped. The section name s is produced by &pick if a primitive in the block is clicked on while attribute “pick=on”. WSection(w) marks the end of a preceding section. WSection() blocks may be nested.

**WSync(w, s)** : w  
synchronize with window system server

WSync(w, s) synchronizes the program with the server attached to window w on those window systems that employ a client-server model. Output to the window is flushed, and WSync() waits for a reply from the server indicating all output has been processed. If s is "yes", all events pending on w are discarded. WSync() is a no-op on window systems that do not use a client-server model.

### A.7 Preprocessor

Unicon features a simple preprocessor that supports file inclusion and symbolic constants. It is a subset of the capabilities found in the C preprocessor, and is used primarily to support platform-specific code sections and large collections of symbols.

**Preprocessor commands**

Preprocessor directives are lines beginning with a dollar sign. The available preprocessor commands are:
$\textit{define symbol text}$ \hspace{1cm} \textit{symbolic substitution}

All subsequent occurrences of symbol are replaced by the text within the current file. Note that $\textit{define}$ does not support arguments, unlike C.

$\textit{include filename}$ \hspace{1cm} \textit{insert source file}

The named file is inserted into the compilation in place of the $\textit{include}$ line.

$\textit{ifdef symbol}$ \hspace{1cm} \textit{conditional compilation}$

$\textit{ifndef symbol}$ \hspace{1cm} \textit{conditional compilation}$

$\textit{else}$ \hspace{1cm} \textit{conditional alternative}$

$\textit{endif}$ \hspace{1cm} \textit{end of conditional code}$

The subsequent lines of code, up to an $\textit{else}$ or $\textit{endif}$, are discarded unless symbol is defined by some $\textit{define}$ directive. $\textit{ifndef}$ reverses this logic.

$\textit{error text}$ \hspace{1cm} \textit{compile error}$

The compiler will emit an error with the supplied text as a message.

$\textit{line number [filename]}$ \hspace{1cm} \textit{source code line}$

$\#line number [filename]$ \hspace{1cm} \textit{source code line}$

The subsequent lines of code are treated by the compiler as commencing from line number in the file filename or the current file if no filename is given.

$\textit{undef symbol}$ \hspace{1cm} \textit{remove symbol definition}$

Subsequent occurrences of symbol are no longer replaced by any substitute text.

EBCDIC transliterations \hspace{1cm} alternative bracket characters

These character combinations were introduced for legacy keyboards that were missing certain bracket characters.

$\{ for$

$\}$ for $

$\langle for$

$\rangle for$

These character combinations are substitutes for curly and square brackets on keyboards that do not have these characters.

**Predefined symbols**

Predefined symbols are provided for each platform and each feature that is optionally compiled in on some platforms. These symbols include:

<table>
<thead>
<tr>
<th>Preprocessor Symbol</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>_V9</td>
<td>Version 9</td>
</tr>
<tr>
<td>_AMIGA</td>
<td>Amiga</td>
</tr>
</tbody>
</table>
A.8 Execution Errors

There are two kinds of errors that can occur during the execution of an Icon program: runtime errors and system errors. Runtime errors occur when a semantic or logic error in a program results in a computation that cannot perform as instructed. System errors occur
when an operating system call fails to perform a required service.

**Runtime errors**

By default, a runtime error causes program execution to abort. Runtime errors are reported by name as well as by number. They are accompanied by an error traceback that shows the procedure call stack and value that caused the error, if there is one. The errors are listed below to illustrate the kinds of situations that can cause execution to terminate.

The keyword `&error` turns runtime errors into expression failure. When an expression fails due to a converted runtime error, the keywords `&errornumber`, `&errortext`, and `&errorvalue` provide information about the nature of the error.

101 integer expected or out of range
102 numeric expected
103 string expected
104 cset expected
105 file expected
106 procedure or integer expected
107 record expected
108 list expected
109 string or file expected
110 string or list expected
111 variable expected
112 invalid type to size operation
113 invalid type to random operation
114 invalid type to subscript operation
115 structure expected
116 invalid type to element generator
117 missing main procedure
118 co-expression expected
119 set expected
120 two csets or two sets expected
121 function not supported
122 set or table expected
123 invalid type
124 table expected
125 list, record, or set expected
126 list or record expected
140 window expected
141 program terminated by window manager
142 attempt to read/write on closed window
malformed event queue
window system error
bad window attribute
incorrect number of arguments to drawing function
window attribute cannot be read or written as requested
cannot open file
bad file attribute
cannot open socket
string or integer expected
posix header file not included
posix record overridden by global value
directory opened for writing
directory or database invalid as file
invalid mode string
invalid signal
invalid operation to flock/fcntl
invalid procedure type
fdup of closed file
low-level read or select mixed with buffered read
not a network connection
not a UDP socket
invalid protocol name
invalid permission string for umask
database expected
division by zero
remaindering by zero
integer overflow
real overflow, underflow, or division by zero
invalid value
negative first argument to real exponentiation
invalid field name
second and third arguments to map of unequal length
invalid second argument to open
non-ascending arguments to detab/entab
by value equal to zero
attempt to read file not open for reading
attempt to write file not open for writing
input/output error
attempt to refresh &main
external function not found
### A.8. EXECUTION ERRORS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>evaluation stack overflow</td>
</tr>
<tr>
<td>302</td>
<td>memory violation</td>
</tr>
<tr>
<td>303</td>
<td>inadequate space for evaluation stack</td>
</tr>
<tr>
<td>304</td>
<td>inadequate space in qualifier list</td>
</tr>
<tr>
<td>305</td>
<td>inadequate space for static allocation</td>
</tr>
<tr>
<td>306</td>
<td>inadequate space in string region</td>
</tr>
<tr>
<td>307</td>
<td>inadequate space in block region</td>
</tr>
<tr>
<td>308</td>
<td>system stack overflow in co-expression</td>
</tr>
<tr>
<td>401</td>
<td>co-expressions not implemented</td>
</tr>
<tr>
<td>402</td>
<td>program not compiled with debugging option</td>
</tr>
<tr>
<td>500</td>
<td>program malfunction</td>
</tr>
<tr>
<td>600</td>
<td>vidget usage error</td>
</tr>
</tbody>
</table>

### System errors

If an error occurs during the execution of a system function, by default the function fails and keywords &errornumber, &errortext and &errorvalue will be set. This contrasts with runtime errors, which terminate execution by default. Whereas runtime errors can be converted to failure by setting &error, system errors can be converted to a runtime error by setting keyword &syserr to a non-null value.

The complete set of system errors is by definition platform specific. Error numbers above the value 1000 are used for system errors. Many of the POSIX standard system errors are supported across platforms, and error numbers between 1001 and 1040 are reserved for the system errors listed below. Platforms may report other system error codes so long as they do not conflict with existing runtime or system error codes.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Operation not permitted</td>
</tr>
<tr>
<td>1002</td>
<td>No such file or directory</td>
</tr>
<tr>
<td>1003</td>
<td>No such process</td>
</tr>
<tr>
<td>1004</td>
<td>Interrupted system call</td>
</tr>
<tr>
<td>1005</td>
<td>I/O error</td>
</tr>
<tr>
<td>1006</td>
<td>No such device or address</td>
</tr>
<tr>
<td>1007</td>
<td>Arg list too long</td>
</tr>
<tr>
<td>1008</td>
<td>Exec format error</td>
</tr>
<tr>
<td>1009</td>
<td>Bad file number</td>
</tr>
<tr>
<td>1010</td>
<td>No child processes</td>
</tr>
<tr>
<td>1011</td>
<td>Try again</td>
</tr>
<tr>
<td>1012</td>
<td>Out of memory</td>
</tr>
<tr>
<td>1013</td>
<td>Permission denied</td>
</tr>
<tr>
<td>1014</td>
<td>Bad address</td>
</tr>
<tr>
<td>1016</td>
<td>Device or resource busy</td>
</tr>
</tbody>
</table>
1017  File exists
1018  Cross-device link
1019  No such device
1020  Not a directory
1021  Is a directory
1022  Invalid argument
1023  File table overflow
1024  Too many open files
1025  Not a typewriter
1027  File too large
1028  No space left on device
1029  Illegal seek
1030  Read-only file system
1031  Too many links
1032  Broken pipe
1033  Math argument out of domain of func
1034  Math result not representable
1035  Resource deadlock would occur
1036  File name too long
1037  No record locks available
1038  Function not implemented
1039  Directory not empty
Appendix B

The Icon Program Library

The Icon Program Library (IPL) includes literally hundreds of complete programs ranging from simple demonstrations to complex applications. More importantly, it offers hundreds of library modules with over a thousand procedures that can save you a lot of effort. To use these procedures in your own programs, you add link declarations. The IPL is generally unpacked in its own directory hierarchy. Unicon knows the location of the IPL hierarchy relative to the Unicon binaries. If you move these binaries or wish to add your own directories of reusable code modules, you must set your IPATH (and often your LPATH) environment variables so they know where to find the IPL directories, in addition to any of your own you wish to add. Both IPATH and LPATH are space-separated lists of directories. The names IPATH and LPATH may be misleading; IPATH directories are searched for precompiled u-code modules with .u extensions, while LPATH directories are searched for $include’d source code files with the .icn extension.

The Icon Program Library has 20 directories of code, data, and supporting documentation. It is included in its entirety in the Unicon distributions. This reference summary appendix presents details for the most useful components of the IPL, emphasizing components of potential value to other programmers. Many modules were written by Icon creator Ralph Griswold; if he was not the author of a given module, the initials of authors are included in the module description. A list of authors’ names appears at the end of the appendix. This appendix no doubt overlooks some excellent library modules and programs. The entries in this appendix loosely follow this template:

filename

Each module has one or more paragraphs of general description, ending with the authors’ initials in parentheses. After the description, if the module consisted of more than a single procedure, the external interface of the module is summarized. The external interface consists of procedures, record types and global variables. The number and type of each procedure’s parameters and return values are given. Library procedures can be expected to return a null value unless otherwise noted. The filename at the beginning is what you link in order to use these procedures in your programs. After the external interface is
summarized, related modules are listed.

record types(fields) with summary descriptions
functions(parameters) : return types with summary descriptions
Links: other library modules (author’s initials)

B.1 Procedure Library Modules

These files appear in the ipl/procs/ directory. Each library module in this directory is a collection of one or more procedures. Some modules also introduce record types or use global variables that are of interest to programmers that use the module.

adlutils
This module processes address lists; addresses are represented using a record type label. Procedures for extracting the city, state, and ZIP code work for U.S. addresses only.

record label(header, text, comments) holds one address
nextadd() : label produces the next address from standard input
writeadd(label) writes an address to standard output
get_country(label) : string produces the country from an address
get_state(label) : string produces the state, or "XX" if no state is found
get_city(label) : string produces the city from an address
get_zipcode(label) : string produces the city from an address
get_lastname(label) : string get last name
get_namepfx(label) : string get name prefix
get_title(label) : string get name title
format_country(string) : string format country name using initial capital letters.
Links: lastname, io, namepfx, title

apply
apply(L:list, argument):any applies a list of functions to an argument. An example is apply([integer,log],10) which is equivalent to integer(log(10)).

argparse
argparse(s:string):list parses s as a command line and returns a list of arguments. At present, it does not accept any escape conventions.

array
This module provides a multidimensional array abstraction with programmer-supplied base indices. create_array(lbs:list, ubs:list, value):array creates an n-dimensional array with specified lower bounds, upper bounds, and with each array element having the given initial value. ref_array(A, i1, i2, ...) references the i1-th i2-th ... element of A.

asciinam
asciiname(s:string):string returns the name of unprintable ASCII character s.
**B.1. PROCEDURE LIBRARY MODULES**

---

**base64** *(DAG)*

This module provides base64 encodings for MIME (RFC 2045). Among other things, this facilitates the writing of programs that read e-mail messages.

- `base64encode(string) : string` returns the base64 encoding of its argument.
- `base64decode(string) : string` returns a base64 decoding of its argument. It fails if the string is not base64 encoded.

**basename** *(REG, CAS)*

`basename(name, suffix) : string` removes any path information as well as the specified suffix, if present. If no suffix is given, the portion of the name up to the first "." is returned.

**binary** *(RJA)*

These procedures support conversion of Icon data elements to and from binary data formats. The control procedures `pack()` and `unpack()` take a format string that controls conversions of several values, similar to the `printf()` C library function.

- `pack(template, value1, ...) : packed_binary_string` packs the value arguments into a binary structure, returning a string containing the structure. The elements of any lists in the value parameters are processed individually as if they were spliced into the value parameter list. The `template` characters give the order and type of values, as follows:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ASCII, null padded</td>
</tr>
<tr>
<td>b</td>
<td>bitstring, low-to-high order</td>
</tr>
<tr>
<td>h</td>
<td>hexadecimal string, low-nibble-first</td>
</tr>
<tr>
<td>c</td>
<td>signed char</td>
</tr>
<tr>
<td>s</td>
<td>signed short</td>
</tr>
<tr>
<td>i</td>
<td>signed integer</td>
</tr>
<tr>
<td>l</td>
<td>signed long</td>
</tr>
<tr>
<td>n</td>
<td>short in &quot;network&quot; order (big-endian)</td>
</tr>
<tr>
<td>v</td>
<td>short in VAX order (little endian)</td>
</tr>
<tr>
<td>f</td>
<td>single-precision float in IEEE Motorola format</td>
</tr>
<tr>
<td>e</td>
<td>extended-precision float in IEEE Motorola format 80-bit</td>
</tr>
</tbody>
</table>
| x      | skip forward a byte (null-fill for `pack()`)
| @      | go to absolute position (null-fill if necessary for `pack()`)

- A | ASCII, space padded |
- B | bitstring, high-to-low order |
- H | hexadecimal string, high-nibble-first |
- C | unsigned char |
- S | unsigned short |
- I | unsigned integer |
- L | unsigned long |
- N | long in "network" order (big-endian) |
- V | long in VAX order (little endian) |
- D | double precision floats in IEEE Motorola format |
- E | extended-precision float in IEEE Motorola format 96-bit |
- X | back up a byte |
- U | uuencoded/uudecoded string |

Each letter may be followed by a numeric count. Together the letter and the count make a field specifier. Letters and numbers can be separated by white space that is ignored. Types in [AaBbHh] consume one value from the “value” list and produce a string of the length
given as the field-specifier-count. The other types consume "field-specifier-count" values from the "value" list and append the appropriate data to the packed string.

`unpack(template, string) : value_list` does the reverse of `pack()`: it takes a string representing a structure and expands it out into a list of values. The template has mostly the same format as for `pack()`.

```plaintext
bincvt

These procedures are for processing of binary data read from a file.

`unsigned(s:string) : integer` converts a binary byte string into an unsigned integer.
`raw(s:string) : integer` puts raw bits of characters of string s into an integer. If the size of s is less than the size of an integer, the bytes are put into the low order part of the integer, with the remaining high order bytes filled with zero. If the string is too large, the most significant bytes will be lost -- no overflow detection.
`rawstring(i:integer, size) : string` creates a string consisting of the raw bits in the low order size bytes of integer i.

bitint

`int2bit(i) : string` produces a string with the bit representation of i.
`bit2int(s) : integer` produces an integer corresponding to the bit representation i.

bitstr,bitstrm

These two modules operate on numeric values represented by arbitrarily long strings of bits, stored without regard to character boundaries. In conjunction with arbitrary precision integers, this facility can deal with bit strings of arbitrary size. record BitString(s, buffer, bufferBits) represents bit strings internally. See the header comments atop the source code for the public API and examples for these modules.

bufread

These procedures provide lookahead within an open file. The procedures `bufopen()`, `bufread()`, and `bufclose()` mirror the built-in `open()`, `read()`, and `close()`.

`bufopen(s:&input): file?` opens a file name s for buffered read and lookahead.
`bufread(f:&input): string` reads the next line from file f. You cannot `bufread() &input` unless you have previously called `bufopen()` on it.
`bufnext(f:&input, n:1): string` returns the next n-th line from file f without changing the next record to be read by `bufread()`.
`bufclose(f:&input): file` close file f.

In addition to processing the current line, one may process subsequent lines before they are logically read. Example:

```plaintext
file := bufopen("name", "r") | stop('open failed')
while line := bufread(file) do {
    ...process current line...
    line := bufnext(file,1) # return next line
    ...process next line...
```
B.1. PROCEDURE LIBRARY MODULES

```plaintext
line := bufnext(file,2) # return 2nd next line
...process 2nd next line... ...etc...
}
bufclose(file)
```

In the code above, calls to `bufnext()` do not affect subsequent `bufread()`’s.

calls

 Procedures to deal with procedure invocations encapsulated in records.
- **record call(proc, args)** encapsulates a procedure and its argument list.
- **invoke(call)** invokes a procedure with an argument from a call record.
- **call_image(call)** produces a string image of a call.
- **make_call(string)** makes a call record from a string image of an invocation.
- **make_args(string)** makes an argument list from a comma-separated string.
- **call_code(string)** produces a string of Icon code to construct a call record.
- **write_calltable(T:table, p:procedure, f:file):null** writes a table of calls (all to procedure p) out to a file. The format is `name=proc:arg1,arg2,?,argn`, 
- **read_calltable(f:file) : table** reads a call table file into a table.

Links: ivalue, procname.

capture

- **capture(f:file)** replaces `write()`, `writes()`, and `stop()` with procedures that echo those elements that are sent to `&output` to the file f.
- **uncaptured_write(s:string...)**, **uncaptured_writes(s:string...)** and **uncaptured_stop(s:string...)** allow output to be directed to `&output` without echoing to the capture file. These are handy for placing progress messages and other comforting information on the screen.

caseless

 These procedures are analogous to the standard string-analysis functions except that uppercase letters are considered equivalent to lowercase letters. They observe the string scanning function conventions for defaulting of the last three parameters.
- **anyccl(c, s, i1, i2)** succeeds and produces `i1 + 1`, provided `map(s[i1])` is in `cset(map(c))` and `i2` is greater than `i1`. It fails otherwise.
- **balccl(c1, c2:'(', c3:')', s, i1, i2)** generates the sequence of integer positions in s preceding a character of `cset(map(c1))` in `map(s[i1:i2])` that is balanced with respect to characters in `cset(map(c2))` and `cset(map(c3))`, but fails if there is no such position.
- **findccl(s1, s2, i1, i2)** generates the sequence of integer positions in s2 at which `map(s1)` occurs as a substring in `map(s2[i1:i2])`, but fails if there is no such position.
- **manyccl(c,s,i1,i2)** produces the position in s after the longest initial sequence of characters in `cset(map(c))` within `map(s[i1:i2])`. It fails if `map(s[i1])` is not in `cset(map(c))`.
- **matchccl(s1, s2, i1, i2) : integer?** produces `i1 + *s1` if `map(s1) == map(s2[i1+:=*s1])`. 
uptocl(c, s, i1, i2) generates the sequence of integer positions in s preceding a character of cset(map(c)) in map(s[i1:i2]). It fails if there is no such position.

cgi

The cgi library provides support for development of Common Gateway Interface server side web based applications, commonly called CGI scripts.

global cgi : table contains keys that are the names of input fields in the invoking HTML page’s form, and values are whatever the user typed in those input fields.

cgiInput(type, name, values) writes HTML INPUT tags with the given type and name for each of a list of values. The first value’s input tag is CHECKED.

cgiSelect(name, values) writes an HTML SELECT with a given name and OPTION tags for a list of values. The first value’s OPTION tag is SELECTED.

cgiXYCoord(hlst) : string is used with an ISMAP. If the x and y coordinates are between certain boundaries, it returns the value of the list element that was entered.

cgiMyURL() : string returns the URL for the current script, as obtained from the SERVER_NAME and SCRIPT_NAME environment variables.

cgiPrintVariables(T) prints a Unicon table using simple HTML formatting.

cgiError(L) generates an error message consisting of the strings in list L, with L[1] as the title and subsequent list elements as paragraphs.

cgiHexVal(c) produces a value from 0 to 15 corresponding to hex chars 0 to F.

cgiHexChar(c1,c2) produces a char corresponding to two char-encoded hex digits.

cgiColorToHex(s) : string produces a 24-bit hex color value corresponding to a string color name. At present, only the colors black, gray, white, pink, violet, brown, red, orange, yellow, green, cyan, blue, purple, and magenta are supported.

cgiPrePro(filename, def) copies out parts of a named HTML file, writing out anything between pairs of <!−− ALL or <!−− def comments.

cgiRndImg(L, s) writes an HTML IMG tag for a random element of L, which should be a list of image filenames. The tag has ALT text given in string s.

cgiOptwindow(opts, args...) : window? attempts to open an Icon window, either on the X server or else on display :0 of the client’s machine (as defined by the IP address in REMOTE_ADDR). The Icon window is typically used to generate a .GIF image to which a link is embedded in the CGI program’s output.

main(args) is included in the CGI library; you do not write your own. The CGI main() procedure generates an HTML header, parses the CGI input fields into a global table cgi, generates a background by calling the user’s cgiBBuilder() function, if any, and calls the user’s cgimain() function.

cgiBBuilder(args...) : table is an optional procedure that a CGI program can use to define the general appearance of its generated web page output. If the user application defines this function, it should return a table which contains keys "background", "bgcolor", "text", 'link', "vlink", "bgproperties" with appropriate values to go into the BODY tag and define background color and texture for the CGI page.
**B.1. PROCEDURE LIBRARY MODULES**

**cgimain(args)** is the entry point for CGI programs. When you use the CGI library, its **main()** initializes things and then calls your **cgimain()** to generate the HTML content body for the client’s web page.

**codeobj**

This module provides a way of storing Icon values as strings and retrieving them.

- **encode(x:any) : string** converts x to a string s.
- **decode(s:string) : any** converts a string in **encode()** format back to x.

These procedures handle all Unicon values, including structures of arbitrary complexity. For scalar types (null, integer, real, cset, and string), \( \text{decode(encode(x))} = x \). For structures (list, set, table, and record types) \( \text{decode(encode(x))} \) is not identical to x, but it has the same "shape" and its elements bear the same relation to the original as if they were encoded and decode individually. Not much can be done with files, functions and procedures, and co-expressions except to preserve type and identification. The encoding of strings and csets handles characters in a way that is safe to write to a file and read it back.

Links: escape, gener, procname, typecode. See also: object.icn.

**colmize**

- **colmize(L:list,mx:80,sp:2,mi:0,tag,tagsp:2,tagmi:0,roww,dist) : string** arranges data items (from list of strings L) into multiple columns. mx is the maximum width of output lines. sp is the minimum number of spaces between columns. mi is the minimum column width. tag is a label to be placed on the first line of output. Items are arranged column-wise unless roww is nonnull; by default the sequence runs down the first column, then down the second, etc. colmize() prints the items in as few vertical lines as possible.

**complete**

- **complete(s:string,st) : string** lets you supply a partial string, s, and get back those strings in st that begin with s. st must be a list or set of strings. Example:

```unicon
commands := ["run","stop","quit","save","load","continue"]
while line := read(&input) do {
  cmds := []
  every put(cmds, complete(line, commands))
  case *cmds of {
    0 : input_error(line)
    1 : do_command(cmds[1])
    default : display_possible_completions(cmds)
  }
}
```

**complex**

The following procedures perform operations on complex numbers.

- **record complex(r,i) creates a complex number with real part r and imaginary part i**
cpxadd(x1,x2) : complex add complex numbers x1 and x2

cpxdiv(x1,x2) : complex divide complex number x1 by complex number x2

cpxmul(x1,x2) : complex multiply complex number x1 by complex number x2

cpxsub(x1,x2) : complex subtract complex numbers x2 from x1

cpxstr(x) : complex convert complex number x to string representation

strcpx(s) : complex convert string to complex number

conffile

This module parses configuration files into Icon structures for easy access. The service is similar to command-line option handling, except that configuration files can contain structured data such as lists or tables. A configuration file supplies values for a set of named directives. The directives and their values are read into a table, usually during program initialization. The types of all allowed directives are specified as follows, before the configuration file is accessed.

**Directive** (classproc, editproc, minargs, maxargs): DirectivesRec produces records (whose field names match the parameter names) that populate a table used to control configuration file processing. **classproc** is a procedure that indicates what kind of value is associated with a given directive in the configuration file:. **editproc** is an optional type conversion procedure to apply to values. **minargs** and **maxargs** are the minimum and maximum number of arguments to a directive; their defaults depend on the **classproc**.

```
classproc     min, max     classproc     min, max
Directive_value  1, 1       Directive_list  1, inf
Directive_table  2, inf     Directive_set    1, inf
Directive_table_of_sets  2, inf   Directive_ignore  0, inf
Directive_warning   0, inf
```

The following code creates a table of specifications for a case-insensitive label and a list of exactly three color components. The default value for tables passed into ReadDirectives() should be a **Directive** for **Directive_ignore** or **Directive_warning**.

```
tos := table(Directive(Directive_ignore))
tos["mylabel"] := Directive(Directive_value, map)
tos["fgrgb"] := Directive(Directive_list, 3, 3)
```

**ReadDirectives** (L:list, T:table, s1:"#", s2:"-", c1:\\, c2:\b\t\v\f\r,p) : table returns a table containing directives parsed and extracted from a file. **L** is a list of string filenames or open files; the procedure uses the first one that it is able to read. **T** is a table of specifications for parsing and handling directives. **s1** is the comment character. **s2** is the continuation character. **c1** are escape characters. **c2** are white space. The following code reads directives using the previously-defined specification table, and creates a window whose title is taken from the specified **mylabel** directive if it is present. Links: lastc.

```
cfg := ReadDirectives([".mycfg", "my.cfg", &input], tos)
WOpen("label=" || (\ (cfg["mylabel"])) | "untitled")
```
B.1. PROCEDURE LIBRARY MODULES

convert

This module contains numeric conversions between bases. There are several other procedures related to conversion that are not yet part of this module.

\textbf{exbase10}(i, j) : string converts base-10 integer \textit{i} to base \textit{j}.

\textbf{inbase10}(s, i) : integer converts base-\textit{i} integer \textit{s} to base 10.

\textbf{radcon}(s, i, j) : integer converts base-\textit{i} integer \textit{s} to base \textit{j}.

created

\textbf{created}(string) : integer returns (approximately) the number of structures of a given type that have been created. Links: \textit{serial}.

currency

\textbf{currency}(\textit{amt}, \textit{wid}:0, \textit{neg}:"-", \textit{frac}:2, \textit{whole}:1, \textit{sign}:"\$", \textit{decimal}:".", \textit{comma}:",") : string formats \textit{amt} in a currency format that defaults to U.S. currency. \textit{amt} can be a real, integer, or numeric string. \textit{wid} is the output field width; its amount is right-adjusted. The returned string will be longer than \textit{width} if necessary to preserve significance. \textit{neg} is the character string to be used for negative amounts, and is placed to the right of the amount. \textit{frac} and \textit{whole} are the exact number of digits to use right of the decimal, and the minimum number of digits to appear left of the decimal, respectively. The currency \textit{sign} prefixes the returned string. The characters used for decimal point and comma may also be supplied.

datecomp

These procedures do simple date comparisons. The parameters are strings of the form mm/dd/yyyy or \&date-compatible yyyy/mm/dd.

\textbf{dgt}(\textit{date1}:string, \textit{date2}:string) : ? succeeds if \textit{date1} is later than \textit{date2}.

\textbf{dlt}(\textit{date1}:string, \textit{date2}:string) : ? succeeds if \textit{date1} is earlier than \textit{date2}.

\textbf{deq}(\textit{date1}:string, \textit{date2}:string) : ? succeeds if \textit{date1} is equal to \textit{date2}.

\textbf{futuredate}(\textit{date}:string) : ? succeeds if \textit{date} is in the future.

\textbf{pastdate}(\textit{date}:string) : ? succeeds if \textit{date} is in the past.

\textbf{getmonth}(\textit{date}:string) : string returns the month portion of a date.

\textbf{getday}(\textit{date}:string) : string returns the day portion of a date.

\textbf{getyear}(\textit{date}:string) : string returns the year portion of a date.

datefns

Date and calendar adaptations of C functions from "The C Programming Language" (Kernighan and Ritchie, Prentice-Hall) and "Numerical Recipes in C" (Press et al, Cambridge). They represent dates using the record type below.

\textbf{record date_rec(year, month, day, yearday, monthname, dayname)}

\textbf{initdate}() initializes the global data before using the other functions.

\textbf{today}() : \textbf{date_rec} produces a computationally useful value for today’s date

\textbf{julian}(\textit{date}) : \textbf{integer} converts a \textit{date} to a Julian day number

\textbf{unjulian}(\textit{julianday}) : \textbf{date} produces a date from the Julian day number
doy(year, month, day) : integer returns the day-of-year from (year, month, day)

wrdate(leadin, date) writes a basic date string preceded by a leadin to &output

datetime

Miscellaneous date and time operations. See also: datefn.icn.

global DateBaseYear : 1970 is a time origin for several functions. An environment variable by the same name overrides the default value.

ClockToSec(string) : integer converts a time in the format of &clock into a number of seconds past midnight.

DateLineToSec(dateline, hoursFromGmt) : integer converts a date in &dateline format to seconds since start of DateBaseYear.

DateToSec(string) : integer converts a date string in Icon &date format (yyyy/mm/dd) to seconds past DateBaseYear.

SecToClock(integer) : string converts a number of seconds past midnight to a string in the format of &clock.

SecToDate(integer) : string converts a number of seconds past DateBaseYear to a string in Icon &date format (yyyy/mm/dd).

SecToDateLine(sec, hoursFromGmt) : string yields a date in &dateline format.

SecToUnixDate(sec, hoursFromGmt) : string returns a date and time in typical UNIX format: Jan 14 10:24 1991.

calendar(j) : date1 returns a record with the month, day, and year corresponding to the Julian Date Number j.

date() : string produces the natural date in English.

dayofweek(day, month, year) : string yields the day of the week for a given date.

full13th(year1, year2) generates records giving the days on which a full moon occurs on Friday the 13th in the range from year1 through year2.

julian(m,d,y) yields the Julian Day Number for the given month, day, and year.

pom(n, phase) returns record with the Julian Day number of fractional part of the day for which the nth such phase since January, 1900. Phases are encoded as:

0 = new moon  1 = first quarter  2 = full moon  3 = last quarter

GMT is assumed.

saytime() computes the time in natural English. If an argument is supplied it is used as a test value to check the operation the program.

walltime() : integer produces the number of seconds since midnight. Beware wrap-around when used in programs that span midnight.

db

These procedures provide an interface to the ODBC database facilities that does not require knowledge of SQL. It also provides compatibility procedures for an earlier version of the ODBC interface.

dbdelete(db, filter...):integer deletes rows from db that satisfy filters. Warning: if the
B.1. PROCEDURE LIBRARY MODULES

filter criteria are omitted, the database will be emptied by this operation!

**dbinsert** *(db, row:record)* inserts a record as a tuple *(row)* into db.

**dbselect** *(db, columns, condition, ordering):integer* selects columns from db. columns defaults to "all", condition defaults to unconditionally, and ordering defaults to unordered.

**dbupdate** *(db:database, row:record)* updates the db tuple corresponding to row.

**dbopen** *(dsn, tabl, user, password):f* is an alias for open*(dsn, "o", ...)*. tabl is optional.

**dbclose** *(db)* is an alias for close()

**dbfetch** *(db)* is an alias for fetch()

**dbsql** *(db, query)* is an alias for sql()

<table>
<thead>
<tr>
<th>dif</th>
<th>(RJA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dif**(strm:list, compare:&quot;==&quot;, eof, group:groupfactor):list**</td>
<td>generates differences between input streams. Returns a list of records, one for each input stream, with each record containing a list of items that differ and their positions in the input stream. The record type is as: record <em>diff_rec</em>(pos, diffs). dif() fails if there are no differences.</td>
</tr>
</tbody>
</table>

strm is a list of input streams from which dif() will extract its input "records". The elements can be any of the following types, with corresponding actions:

<table>
<thead>
<tr>
<th>Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>file is &quot;read&quot; to get records</td>
</tr>
<tr>
<td>co-expression</td>
<td>co-expression is activated to get records</td>
</tr>
<tr>
<td>list</td>
<td>records are &quot;gotten&quot; (get()) from the list</td>
</tr>
<tr>
<td>diff_proc</td>
<td>a record type that has two fields, a procedure to call and the argument to pass to it. Its definition is: record <em>diff_proc</em>(proc, arg). It allows procedures supplied by dif's caller to be called to get records.</td>
</tr>
<tr>
<td>compare</td>
<td>is a procedure that succeeds if two records are &quot;equal&quot;, and fails otherwise. The comparison must allow for the fact that the EOF object might be an argument, and a pair of EOFs must compare equal.</td>
</tr>
<tr>
<td>eof</td>
<td>is an object that is distinguishable from other objects in the stream.</td>
</tr>
<tr>
<td>group</td>
<td>is a procedure that is called with the current number of unmatched items as its argument. It must return the number of matching items required for file synchronization to occur. The default (procedure groupfactor()) is the formula Trunc((2.0 * Log(M)) + 2.0) where M is the number of unmatched items.</td>
</tr>
</tbody>
</table>

digitcnt

digitcnt**(file:&input)** : list counts the number of each digit in a file and returns a ten-element list with the counts.

equiv

equiv**(x,y)** : any? tests the equivalence of two values. For non-structures, it returns x1 === x2. For structures, the test is for shape. For example, equiv([[]], [[]]) succeeds. It handles loops, but does not recognize them as such. The concept of equivalence for tables and sets is weak if their elements are themselves structures. There is no concept of order for tables and sets, yet it is impractical to test for equivalence of their elements without imposing
an order. Since structures sort by "age", there may be a mismatch between equivalent structures in two tables or sets.

---

**escapes**

These procedures manipulate escape sequences in Icon string format.

- **escapeseq() : string** is a matching procedure for Icon string escape sequences
- **escchar(string) : string** produces the character value of an Icon escape sequence
- **escape()** converts a string with escape sequences to the string it represents. For example, **escape("\\143\\141\\164")** produces the string "cat".
- **quotedstring()** matches a complete quoted string.

---

**eval**

- **eval(string) : any** analyzes a string representing an Icon function or procedure call and evaluates the result. Operators can be used in functional form, as in "*(2,3)". This procedure cannot handle nested expressions or control structures. It assumes the string is well formed. The arguments can only be Icon literals. Escapes, commas, and parentheses in string literals are not handled. In the case of operators that are both unary and binary, the binary form is used. Links: *ivalue*.

---

**evallist**

- **evallist(expr, n, ucode, ...) : list** returns a list of the results written by a program consisting of expression **expr** (normally a generator); **n** is the maximum size of the list, and the trailing arguments are ucode files to link with the expression. Requires: **system()**, */tmp*, pipes. Links: *exprfile*.

---

**everycat**

- **everycat(x1, x2, ...) : string** generates the concatenation of every string from !x1, !x2, ... For example, if first := ["Mary", 'Joe', "Sandra"] and last := ["Smith", "Roberts"] then **every write(everycat(first, " ", last))** writes Mary Smith, Mary Roberts, Joe Smith, Joe Roberts, Sandra Smith, Sandra Roberts. x1, x2, ... can be any values for which !x1, !x2, ... are convertible to strings. In the example, the second argument is a one-character string " ", so that !" ' generates a single blank.

---

**exprfile**

- **exprfile(exp, link, ...) : file** produces a pipe to a program that writes all the results generated by **exp**. The trailing arguments name link files needed for the expression. **exprfile()** closes any previous pipe it opened and deletes its temporary file. Therefore, **exprfile()** cannot be used for multiple expression pipes. If the expression fails to compile, the global **expr_error** is set to 1; otherwise 0.
- **exec_expr(expr_list, links[]) : string** generates the results of executing the expression contained in the lists **expr_list** with the specified links.

Requires: **system()**, pipes, */tmp*. Links: io.

---

**factors**

(REG, GMT)
This file contains procedures related to factorization and prime numbers.

- **factorial(n)** returns n!. It fails if n is less than 0.
- **factors(i, j)** returns a list containing the factors of i, up to maximum j; by default there is no limit.
- **gfactorial(n, i)** generalized factorial; n x (n - i) x (n - 2i) x ...
- **ispower(i, j)** succeeds and returns root if i is k \textasciitilde j
- **isprime(n)** succeeds if n is a prime.
- **nxtprime(n)** returns the next prime number beyond n.
- **pfactors(i)** returns a list containing the primes that divide i.
- **prdecomp(i)** returns a list of exponents for the prime decomposition of i.
- **prime()** generates the primes.
- **primel()** generates the primes from a precompiled list.
- **primorial(i,j)** product of primes j \leq i; j defaults to 1.
- **sfactors(i, j)** is the same as **factors(i, j)**, except output is in string form with exponents for repeated factors

Requires: Large-integer arithmetic and prime.lst for **primel()**. Links: io, numbers.

---

**fastfncs**

These procedures implement integer-values using a best known method.

- **acker(i, j)** Ackermann’s function
- **fib(i)** Fibonacci sequence
- **g(k, i)** Generalized Hofstadter nested recurrence
- **q(i)** "Chaotic" sequence
- **robbins(i)** Robbins numbers

See also: iterfncs.icn, recrfncs.icn. Links: factors, memrfncs.

---

**filedim**

- **filedim(s, p)** : textdim computes the number of rows and maximum column width of the file named s. The procedure p, which defaults to detab, is applied to each line. For example, to have lines left as is, use **filedim(s, 1)**. The return value is a record that uses the declaration record textdim(cols, rows).

---

**filenseq**

- **nextseqfilename(dir, pre, ext)** : string? creates the next filename in a series of files (such as successive log files). Usage: fn := nextseqfilename(".", ".", "log") returns the (non-existent) filename next in the sequence .\*.*.log (* represents 1, 2, 3, ...) or fails.

---

**findre**

- **findre(s1,s2,i,j)** : integer* is like the built-in function **find()**, except its first argument is a regular expression similar to the ones the Unix egrep command uses. A no argument invocation wipes out all static structures utilized by **findre()**, and then forces a garbage collection. **findre()** offers a simple and compact wildcard-based search system. If you do many searches through text files, or write programs which do searches based on user input,
then `findre()` is a utility you might want to look over. `findre()` leaves the user with no easy way to tab past a matched substring, as with \texttt{s ? write(tab(find("hello")+5))}. In order to remedy this intrinsic deficiency, `findre()` sets the global variable `__endpoint` to the first position after any given match occurs. `findre()` utilizes the same basic language as `egrep`, but uses intrinsic Icon data structures and escaping conventions rather than those of any particular Unix variant. `findre()` takes a shortest-possible-match approach to regular expressions. In other words, if you look for "a*", `findre()` will not even bother looking for an "a". It will just match the empty string.

- ^ - matches if what follows is at the beginning of a line
- $ - matches if the preceding pattern is at the end of a line
- . - matches any single character
- + - matches 1 or more occurrences of the previous expression (i.e. a character, or set of parenthesized/bracketed characters)
- * - matches from 0 to any number of occurrences of the previous expression
- \ - removes the special meaning of any special characters
  (i.e. if you want to match lines beginning with a "[", write \[, and not ^[]
- | - matches either the pattern before it, or the one after it
- [] - matches any member of the enclosed character set, or, if ^ is the first character, any nonmember of the enclosed character set
- () - used for grouping (e.g. `(abc|cde)` matches lines consisting of either "abc" or "cde", while "abc|cde" matches lines either beginning with "abc" or ending in "cde")

\begin{verbatim}
\texttt{gauss} \hspace{1cm} (SBW)
\texttt{gauss_random(x, f)} produces a Gaussian distribution about the value \texttt{x}. Parameter \texttt{f} can be used to alter the shape of the Gaussian distribution (larger values flatten the curve...) Produce a random value within a Gaussian distribution about 0.0. (Sum 12 random numbers between 0 and 1, (expected mean is 6.0) and subtract 6 to center on 0.0.

\texttt{gdl, gdl2} \hspace{1cm} (RLG)
\texttt{gdl(dir:string) : list} returns a list containing everything in a directory. You can use this file as a template, modifying the procedures according to the needs of the program in which they are used.
\texttt{gdlrec(dir, findflag) : list} does same thing as \texttt{gdl} except it recursively descends through subdirectories. If \texttt{findflag} is nonnull, the UNIX "find" program is used; otherwise the "ls" program is used.

\texttt{gener} \hspace{1cm}
These procedures generate sequences of results.
\texttt{days()} : \texttt{string} produces the days of the week, starting with "Sunday".
\texttt{hex()} : \texttt{string} is the sequence of hexadecimal codes for numbers from 0 to 255
\texttt{label(s,i)} : \texttt{string} produces labels with prefix \texttt{s} starting at \texttt{i}
\texttt{multii(i, j)} : \texttt{integer} produces \texttt{i * j} i's
months() : string* produces the months of the year
octal() : string* produces the octal codes for numbers from 0 to 255
star(s) : string* produces the closure of s starting with the empty string and
continuing in lexical order as given in s

genrfncs

These procedures generate various mathematical sequences of results. Too many are
included to list them all here; consult the source code for a complete listing.

arithseq(i, j) : string* arithmetic sequence starting at i with increment j.
chaosseq() chaotic sequence
factseq() factorial sequence
fibseq(i, j, k) generalized Fibonacci (Lucas) sequence with additive constant k
figurseq(i) series of i'th figurate number
fileseq(s, i) generate lines (if i is null) or characters (except line terminators)
from file s.
geomseq(i, j) geometric sequence starting at i with multiplier j
irepl(i, j) j instances of i
multiseq(i, j, k) sequence of (i * j + k) i's
ngonalseq(i) sequence of the i polygonal number
primeseq() the sequence of prime numbers
powerseq(i, j) sequence i ^ j, starting at j = 0
spectseq(r) spectral sequence integer(i * r), i - 1, 2, 3, ...
starseq(s) sequence consisting of the closure of s starting with the empty string
and continuing in lexical order as given in s

Requires: co-expressions. Links: io, fastfn cs, partit, numbers.

getmail(x):message_rec* reads an Internet mail folder and generates a sequence of
records, one per message, failing when end-of-file is reached. Each record contains the
message header and message text components parsed into fields. The argument x is either
the name or the file handle. If getmail() is resumed after the last message is generated, it
closes the mail folder and returns failure. If getmail() generates an incomplete sequence
(does not close the folder and return failure) and is then restarted (not resumed) on the
same or a different mail folder, the previous folder file handle remains open and inaccessible.
If message_records are stored in a list, the records may be sorted by individual components
(like sender, _date, _subject) using the sortf() function.

ggetpaths(args[]) : string* generates the paths supplied as arguments followed by those
paths in the PATH environment variable, if one is available. A typical invocation might
look like:

    open(getpaths("/usr/local/lib/icon/procs") || filename)
getpaths() will be resumed in the above context until open succeeds in finding an existing, readable file.

graphpak

The procedures here use sets to represent directed graphs. See "The Icon Programming Language", third edition, pp. 233-236. A graph has two components: a list of nodes and a two-way lookup table. The nodes in turn are sets of pointers to other nodes. The two-way table maps a node to its name and vice-versa. Graph specifications are given in files in which the first line is a white-space separated list of node names and subsequent lines give the arcs, as in

Tucson Phoenix Bisbee Douglas Flagstaff
Tucson->Phoenix
Tucson->Bisbee
Bisbee->Bisbee
Bisbee->Douglas
Douglas->Phoenix
Douglas->Tucson
record graph(nodes:list, lookup:table) represents a graph
read_graph(f:file) : graph reads a graph from a file
write_graph(g:graph, f:file) : null writes a graph to a file
closure(node) : set computes the transitive closure of a node.

hexcvt

hex(s) : integer converts a string of hex digits into an integer.
hexstring(i,n,lc) : string produces the hexadecimal representation of the argument. If n is supplied, a minimum of n digits appears in the result; otherwise there is no minimum, and negative values are indicated by a minus sign. If lc is non-null, lowercase characters are used instead of uppercase.

html

These procedures assist in processing HTML files:
htchunks(file) : string* generates HTML chunks in a file. Results beginning with <!-- are unclosed comments (legal comments are deleted); < begins tags; others are untagged text.
htrefs(f) : string* generates the tagname/keyword/value combinations that reference other files. Tags and keywords are returned in upper case.
httag(string) : string produces the name of a tag contained within a tag string.
htvals(s) : string* generates the keyword/value pairs from a tag.
urlmerge(base,new) : string interprets a new URL in the context of a base URL.

ichartp

ichartp implements a simple chart parser - a slow but easy-to-implement strategy for parsing context free grammars (it has a cubic worst-case time factor). Chart parsers are
flexible enough to handle a variety of natural language constructs, and lack many of the troubles associated with empty and left-recursive derivations. To obtain a parse, create a BNF file, obtain a line of input, and then invoke parse_sentence(sentence, bnf_filename, start-symbol). Parse_sentence() suspends successive edge structures corresponding to possible parses of the input sentence. BNF grammars are specified using the same notation used in Griswold & Griswold, and as described in the IPL program "pargen.icn" later in this appendix with the following difference: All metacharacters (space, tab, vertical slash, right/left parentheses, brackets and angle brackets) are converted to literals by prepending a backslash. Comments can be included along with BNF rules using the same notation as for Icon code (i.e. #-sign).

Links: trees, rewrap, scan, strip, stripcom, strings. Requires: co-expressions.

\[
\text{iftrace} \quad \text{(SBW, REG)}
\]

These procedures trace Icon functions by using procedure wrappers to call the functions. iftrace(fnsc[]) sets tracing for a list of function names. Links: ifnsc.

\[
\text{image} \quad \text{(MG, REG, DY)}
\]

\[\text{Image(x, style:}1) : \text{string}\]

generalizes image(x), providing detailed information about structures. The \text{style} determines the formatting and order of processing. Style 1 is indented, with | and ) at end of last item. Style 2 is also indented, with | and ) on new line. Style 3 puts the whole image on one line. Style 4 is like style 3, with structures expanded breadthfirst instead of depthfirst as for other styles.

Structures are identified by a letter identifying the type followed by an integer. The tag letters are "L" for lists, "R" for records, "S" for sets, and "T" for tables. The first time a structure is encountered, it is imaged as the tag followed by a colon, followed by a representation of the structure. If the same structure is encountered again, only the tag is given.

\[
inbits \quad \text{(RLG)}
\]

\[\text{inbits(file, len:integer)} : \text{integer}\]

re-imports data converted into writable form by outbits(). See also: outbits.icn.

\[
\text{indices} \quad \text{(MG, REG, DY)}
\]

\[\text{indices(spec:list, last)} : \text{list}\]

produces a list of the integers given by the specification \text{spec}, which is a comma separated list of positive integers or integer spans, as in "1,3-10,..." If \text{last} is specified, it is used for a span of the form "10-". In a span, the low and high values need not be in order. For example, "1-10" and "10-1" are equivalent. Similarly, indices need not be in order, as in "3-10, 1,...". Empty values, as in "10,12" are ignored. indices() fails if the specification is syntactically erroneous or if it contains a value less than 1.

\[
\text{inserts} \quad \text{(MG, REG, DY)}
\]

\[\text{inserts(table,key,value)} : \text{table}\]

inserts values into a table in which the same key can
have more than one value (i.e., duplicate keys). The value of each element is a list of inserted values. The table must be created with default value &null. (RJA)

\begin{verbatim}
intstr

    intstr(i:integer,size:integer) : string produces a string consisting of the raw bits in the low order size bytes of integer i. This procedure is used for processing of binary data to be written to a file. Note that if large integers are supported, this procedure still will not work for integers larger than the implementation defined word size due to the shifting in of zero-bits from the left in the right shift operation.
\end{verbatim}

\begin{verbatim}
io

    These procedures provide facilities for handling input, output, and files. Some require loadfunc(). Links: random, strings.

    fcopy(fn1:string,fn2:string) copies a file named fn1 to file named fn2.
    exists(name:string) : file? succeeds if name exists as a file but fails otherwise.
    filelist(s,x) : list returns a list of the file names that match the specification s. If x is nonnull, any directory is stripped off. At present it only works for UNIX.
    filetext(f) : list reads the lines of f into a list and returns that list
    readline(file) : string? assembles backslash-continued lines from the specified file into a single line. If the last line in a file ends in a backslash, that character is included in the last line read.
    splitline(file, line, limit) splits line into pieces at first blank after the limit, appending a backslash to identify split lines (if a line ends in a backslash already, that’s too bad). The pieces are written to the specified file.

\end{verbatim}

\begin{verbatim}
Buffered input and output: ClearOut() remove contents of output buffer without writing

    Flush() flush output buffer
    GetBack() get back line written
    LookAhead() look ahead at next line
    PutBack(s) put back a line
    Read() read a line
    ReadAhead(n) read ahead n lines
    Write(s) write a line

    See also module bufread for a multi-file implementation of buffered input.
\end{verbatim}

\begin{verbatim}
Path searching: dopen(s) : file? opens and returns the file s on DPATH.

    dpath(s) : string? returns the path to s on DPATH.
    pathfind(fname, path:getenv("DPATH")) : string? returns the full path of fname if found along the space-separated list of directories "path". As is customary in Icon path searching, "." is prepended to the path.
\end{verbatim}
B.1. PROCEDURE LIBRARY MODULES

pathload(fname,entry) calls loadfunc() to load entry from the file fname found on the function path. If the file or entry point cannot be found, the program is aborted. The function path consists of the current directory, then getenv("FPATH"), and finally any additional directories configured in the code.

Parsing file names:  suffix() : list parses a hierarchical file name, returning a 2-element list: [prefix,suffix]. For example, suffix("/a/b/c.d") produces "/a/b/c",d"

tail() : list parses a hierarchical file name, returning a 2-element list: [head,tail]. For example, tail("/a/b/c.d") produces "/a/b","c.d".

components() : list parses a hierarchical file name, returning a list of all directory names in the file path, with the file name (tail) as the last element. For example, components("/a/b/c.d") produces ["/", "a", "b", "c.d"].

Temporary files:  tempfile(prefix:"",suffix:"",path:".",len:8) produces a temporary file that can be written. The name is chosen so as not to overwrite an existing file. The prefix and suffix are prepended and appended, respectively, to a randomly chosen number. The path is prepended to the file name. The randomly chosen number is fit into a field of len characters by truncation or right filling with zeros as necessary. It is the user’s responsibility to remove the file when it is no longer needed.

tempname(prefix:"", suffix:"", path:".", len:8) returns a temporary file name.

iolib (RLG, NA)

This library provides control functions for text terminals, based on legacy ANSI and VT-100 devices. The TERM and TERMCAP environment variables must be set in order to use this library. TERM tells iolib what driver you are using, e.g. TERM=ansi-color. The TERMCAP variable gives the location of the termcap database file, as provided on your UNIX system.

Some useful termcap codes are "cl" (clear screen), "ce" (clear to end of line), "ho" (go to the top left square on the screen), "so" (begin standout mode), and "se" (end standout mode). The termcap database holds both string- and numeric-valued sequences. The value of "li" tells how many lines the terminal has; "co" tells how many columns. To go to the beginning of the second-to-last line on the screen: iputs(igoto(getval("cm"), 1, getval("li")-1)). The "cm" capability is a special capability, and needs to be output via igoto(cm,x,y), where cm is the sequence telling your computer to move the cursor to a specified spot, x is the column, and y is the row.

setname(term) initializes the terminal. Overrides TERM environment variable.

getval(id) returns the terminal code for the capability named id. Integer valued capabilities are returned as integers, strings as strings, and flags as records (if a flag is set, then type(flag) will return "true"). Absence of a capability is signaled by failure.

igoto(cm,column,row) returns a terminal code string that causes the cursor to move to the designated column and row. cm is the cursor movement command for the current
terminal, as obtained via getval("cm"). column and row count down and to the right starting from the location $(1,1)$.

`puts(s)` sends $s$ out to the console. For example, to clear the screen, `iputs(getval("cl"))`

Requires: UNIX, co-expressions. See also: iscreen.icn.

### iscreen

This file contains some rudimentary screen functions for use with iolib.icn.

- `clear()` clears the screen (tries several methods)
- `clear_emphasize()` clears the screen to all-emphasize mode.
- `emphasize()` initiates emphasized (usually reverse video dark on light) mode
- `boldface()` initiates bold mode
- `blink()` initiates blinking mode
- `normal()` resets to normal mode
- `message(s)` displays message $s$ on 2nd-to-last line
- `underline()` initiates underline mode
- `status_line(s,s2,p)` draws status line $s$ on the 3rd-to-last screen line; if $s$ is too short for the terminal, $s2$ is used; if $p$ is nonnull then it either centers, left-, or right-justifies, depending on the value, "c", "l", or "r".

Requires: UNIX. Links: iolib.

### isort

`isort(x,keyproc,y) : list` is a customizable sort procedure. $x$ can be any Icon data type that supports the unary element generation (!) operator. The result is a sorted list of objects. The sort keys are obtained by calling keyproc() on each element of structure $x$ to obtain the sort key for that element. If keyproc is a procedure, the first argument to each call to keyproc() is the element for which the key is to be computed, and the second argument is isort’s argument $y$, passed unchanged. The `keyproc` must produce the extracted key. Alternatively, `keyproc` can be an integer, in which case it is a subscript applied uniformly to each element to select a sort key. If `keyproc` is omitted, sorting uses the standard Icon sort order.

### itokens

`itokens(file, nostrip) : TOK*` breaks Icon source files up into tokens for use in things like pretty printers, preprocessors, code obfuscators, and so forth. itokens() suspends values of type record TOK(sym:string, str:string). sym contains the name of the next token, such as "CSET", or "STRING". str gives that token’s literal value. For example, the TOK for a literal semicolon is TOK("SEMICOL", ";"). For a mandatory newline, itokens() would suspend TOK("SEMICOL", "\n"). itokens() fails on end-of-file. It returns syntactically meaningless newlines if the second argument is nonnull. These meaningless newlines are returned as TOK records with a null sym field (i.e. TOK(&null, "\n")). If new reserved words or operators are added to a given implementation, the tables in this module have to be altered. Note: keywords are implemented at the syntactic level; they are not tokens. A
keyword like `&features` is suspended as an `&` token followed by an identifier token. Links: `scan`. Requires: co-expressions.

**ivalue**

`ivalue(s):any` turns a string from `image()` into the corresponding Icon value. It can handle integers, real numbers, strings, csets, keywords, structures, and procedures. For the image of a structure, it produces a result of the correct type and size, but values in the structure are not correct, since they are not encoded in the image. For procedures, the procedure must be present in the environment in which `ivalue()` is evaluated. This generally is true for built-in procedures (functions). All keywords are supported. The values produced for non-constant keywords are, of course, the values they have in the environment in which `ivalue()` is evaluated. `ivalue()` handles non-local variables (`image()` does not produce these), but they must be present in the environment in which `ivalue()` is evaluated.

**jumpque**

`jumpque(queue:list,y):list` moves `y` to the head if it is in the queue; otherwise it adds `y` to the head of the queue. A copy of the queue is returned.

**kmap**

`kmap(string) : string` maps uppercase and control characters into the corresponding lowercase letters. It is for graphic applications in which the modifier keys for shift and control are encoded in keyboard events.

**lastc**

These string scanning functions follow standard conventions for defaulting the last three parameters to the current scanning environment. (DAG)

`lastc(c:cset, s:string, i1:integer, i2:integer) : integer` succeeds and produces `i1`, provided either that `i1` is `1`, or that `s[i1 - 1]` is in `c` and `i2` is greater than `i1`.

`findp(c:cset, s1:string, s2:string, i1:integer, i2:integer) : integer*` generates the sequence of positions in `s2` at which `s1` occurs provided that `s2` is preceded by a character in `c`, or is found at the beginning of the string.

`findw(c1:cset, s1:string, c2:cset, s2:string, i1:integer, i2:integer) : integer*` generates the sequence of positions in `s2` at which `s1` occurs provided that `s2` is preceded and followed by the empty string or a member of `c1` and `c2`, respectively.

**lastname**

`lastname(s:string) : string` produces the last name in string `s`, which must be a name in conventional form. Obviously, it doesn’t work for every possibility.

**list2tab**

`list2tab(list) : null` writes a list as a tab-separated string to `&output`. Carriage returns in files are converted to vertical tabs. See also: `tab2list`, `tab2rec`, `rec2tab`.

lists (REG, RLG)
Many of these procedures implement list functions similar to string functions, including an implementation of list scanning, similar to Icon’s string scanning functions.

\[ \text{lcomb}(L, i) : L^n \] produces all sublist combinations of \( L \) that have \( i \) elements.

\[ \text{ldelete}(L, \text{spec} : \text{string}) : L \] deletes values of \( L \) at indices given in \( \text{spec} \); see \textit{indices}.

\[ \text{lequiv}(L_1, L_2) : L? \] tests if \( L_1 \) and \( L_2 \) are structurally equivalent. 

\[ \text{lequiv}() \] does not detect cycles.

\[ \text{lextend}(L, i, x) : L \] extends \( L \) to at least size \( i \), using initial value \( x \).

\[ \text{limage}(L) : \text{string} \] list image function that shows elements’ images, one level deep.

\[ \text{linterl}(L_1, L_2) : L \] interleaves elements of \( L_1 \) and \( L_2 \). If \( L_1 \) and \( L_2 \) are unequal size, the shorter list is extended with null values to the size of the larger list.

\[ \text{llpad}(L, i, x) : L \] produces a new list that extends \( L \) on its front (left) side.

\[ \text{lltrim}(L, S) : L \] produces a copy of \( L \) with elements of \( S \) trimmed on the left.

\[ \text{lmap}(L_1, L_2, L_3) : L \] maps elements of \( L_1 \) according to \( L_2 \) and \( L_3 \), similar to the string-mapping function \textit{map}(s1, s2, s3). Elements in \( L_1 \) that are the same as elements in \( L_2 \) are mapped into the corresponding elements of \( L_3 \). For example, given the lists

\[ L_1 := [1, 2, 3, 4]; L_2 := [4, 3, 2, 1]; L_3 := ["a", "b", "c", "d"] \]

\( \text{lmap}(L_1, L_2, L_3) \) produces a new list \["d", "c", "b", "a"]\). The lists can have any kinds of elements. The operation \( x === y \) is used to determine if elements \( x \) and \( y \) are equivalent. No defaults are provided for omitted arguments.

As with \text{map()}, \text{lmap()}, can be used for transposition as well as substitution. Warning: If \text{lmap()} is called with the same lists \( L_2 \) and \( L_3 \) as in the immediately preceding call, the same mapping is performed, even if the values in \( L_2 \) and \( L_3 \) have been changed. This improves performance, but it may cause unexpected effects.

\[ \text{lpalin}(L) : L \] produces a list palindrome of \( L \) concatenated with its reverse.

\[ \text{lpermute}(L) : L^n \] produces all the permutations of list \( L \).

\[ \text{lreflect}(L, i : 0) : L \] returns \( L \) concatenated with its reversal to produce a palindrome. The values of \( i \) determine "end conditions" for the reversal:

\[ 0 = \text{omit first and last elements}. \quad 1 = \text{omit first element}. \quad 2 = \text{omit last element}. \quad 3 = \text{don’t omit element}. \]

\[ \text{lremvals}(L, x_1, x_2, \ldots) : \text{list} \] produces a copy of \( L \), without those elements that are \( === x_1, x_2 \)?

\[ \text{lrep}(L, i) : L \] replicates \( L \) \( i \) times.

\[ \text{lrotate}(L, i) : L \] produces a list with the elements of \( L \), rotated \( i \) positions.

\[ \text{lrpad}(L, i, x) : L \] is like \text{lextend}(), but produces a new list instead of changing \( L \).

\[ \text{lrtrim}(L, S : \text{set}) : L \] produces a copy of \( L \) with elements of \( S \) trimmed on the left.

\[ \text{lswap}(L) : L \] produces a copy of \( L \) with odd elements swapped with even elements.

\[ \text{lunique}(L) : L \] produces a list containing only unique list elements.
List Scanning   global l_POS, l_SUBJ are the current list scanning environment
   record l_ScanEnvir(subject, pos) represents (nested) list scanning environments
   l_Bscan(e1) : l_ScanEnvir enter (possibly nested) list scanning environment
   l_Escan(l_OuterEnvir, e2) : any exit list scanning environment

List Scanning Function  String Scanning Equivalent
l_any(l1, l2, i, j) : integer?  any()
l_bal(l1, l2, l3, l, i, j) : integer*  bal()
l_find(l1, l2, i, j) : integer*  find()
l_many(l1, l2, i, j) : integer?  many()
l_match(l1, l2, i, j) : integer?  match()
l_move(i) : list  move()
l_pos(i) : integer  pos()
l_tab(i) : list  tab()
l_upto(l1, l2, i, j) : integer?  upto()

This set of functions implements operations for analyzing lists of tokens similar to string
scanning. l_find() and l_match() accept lists as their first argument. l_any(), l_many(),
and l_upto() take either sets of lists or lists of lists. l_bal() has no defaults for the first four
arguments, since there is no precise list analogue to &cset, etc. List scanning environments
are not maintained as elegantly as for string scanning. You must use a set of nested
procedure calls l_Bscan() and l_Escan(), as explained in the Icon Analyst 1:6 (June, 1991),
p. 1-2. In particular, you cannot suspend, return, or otherwise break out of the nested
procedure calls; they can only be exited via failure. Here is an example of how list scanning
might be invoked:

    suspend l_Escan(l_Bscan(some_list_or_other), {  
        l_tab(10 to *l_SUBJ) & {  
            if l_any(l1) | l_match(l2) then old_l_POS + (l_POS-1)  
        }  
    })

List scanning environments may be nested, and may be used to generate a series of
results as well. Here’s an example of nested list scanning:

procedure main()
    l := ["h", "e", "l", "l", "o", " ", "t", "t", "h", "e", "r", "e"]
    l_Escan(l_Bscan(l), {  
        hello_list := l_tab(l_match(["h", "e", "l", "l", "o"]))
        every writes(!hello_list)
        write() # Note the nested list-scanning expressions.
    })
    l_Escan(l_Bscan(l_tab(0)), {  
        l_tab(l_many([" "], ["t"]))  
        every writes(!l_tab(0))
        write()  
    })

This program writes "hello" and "there" on successive lines to the standard output.

The functions compare lists, not strings, so \_find("h", l) for instance, will yield an error message: use \_find(["h"], l) instead. This becomes confusing when looking for lists within lists. Suppose \_l1 := ["junk", ["hello"], ",," ["there"], "!","m","o","r","e","junk"] and you wish to find the position in \_l1 at which the list ["hello", ",," ["there"] occurs. If you assign \_l2 := [""hello"", ",," ["there"]], then the \_find() call needs to look like \_find([\_l2],\_l1). Links: \_indices.

loadfile

loadfile(exp, link,...) creates and loads a program that generates the results of exp. The trailing arguments name link files needed for the expression. loadfile() returns a procedure that generates the results. Requires: system(), pipes, /tmp. Links: io.

longstr

String scanning function longstr(l,s,i,j) : integer? works like any(), except that instead of taking a cset as its first argument, it takes instead a list or set of strings (l). Returns i + \*x, where x is the longest string in l for which match(x,s,i,j) succeeds, if there is such an x.

lrgapprx

lrgapprx(i) : string produces an approximate of an integer value in the form i.jx10^k. It is primarily useful for large integers.

lu

lu_decomp(M, I) : real? performs LU decomposition on the square matrix M using the vector I. Both M and I are modified. The value returned is +1.0 or -1.0 depending on whether the number of row interchanges is even or odd. lu_decomp() is used in combination with lu_back_sub() to solve linear equations or invert matrices. lu_decomp() fails if the matrix is singular.

lu_back_sub(M, I, B) solves the set of linear equations M x X = B. M is the matrix as modified by lu_decomp(). I is the index vector produced by lu_decomp(). B is the right-hand side vector and return with the solution vector. M and I are not modified by lu_back_sub() and can be used in successive calls of lu_back_sub() with different Bs. Acknowledgement: These procedures are based on algorithms given in "Numerical Recipes; The Art of Scientific Computing"; W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling; Cambridge University Press, 1986.

mapbit

mapbit(s:string) : string produces a string of zeros and ones corresponding to the bit patterns for the characters of s. For example, mapbit("Axe") produces "0100000101111000001100101". Links: strings.
B.1. PROCEDURE LIBRARY MODULES 419

mapstr

mapstrs(string, l1:list, l2:list) : string works like map(), except that instead of taking ordered character sequences (strings) as arguments 2 and 3, it takes ordered string sequences (lists). Suppose, for example, you wanted to bowdlerize a string by replacing the words "hell" and "shit" with "heck" and "shoot." You would call mapstrs as follows:

mapstrs(s, ["hell", "shit"], ["heck", "shoot"]).

If you want to replace one string with another, just use the IPL replace() routine (in strings.icn). If l2 is longer than l1, extra members in l2 are ignored. If l1 is longer, however, strings in l1 that have no correspondent in l2 are simply deleted. mapstr() uses a longest-possible-match approach, so that replacing ["hellish", "hell"] with ["heckish", "heck"] will work as one would expect. Links: longstr.

math

binocoef(n:integer, k:integer) : integer? produces the binomial coefficient n over k. It fails unless 0 <= k <= n.

cosh(r:real) : real produces the hyperbolic cosine of r.
sinh(r:real) : real produces the hyperbolic sine of r.
tanh(r:real) : real produces the hyperbolic tangent of r.

Links: factors.

matrix

This file contains procedures for matrix manipulation. Matrices (arguments beginning with M) are represented as lists of lists. Links: lu.

matrix_width(M) : integer produces the number of columns in a matrix.

matrix_height(M) : integer produces the number of rows in a matrix.

write_matrix(file, M, x) outputs a matrix to a file, one row per line. If x is nonnull, elements are comma-separated and rows are enclosed in square brackets, like list literals.

copy_matrix(M) : list produces a copy of a matrix.

create_matrix(n,m,x) : list creates an n by m matrix with initial value x.

identity_matrix(n,m) : list produces an identity matrix of size n by m.

add_matrix(M1,M2) : list produces the matrix addition of M1 and M2.

mult_matrix(M1,M2) : list produces the matrix multiplication of M1 and M2.

invert_matrix(M) : list produces the matrix inversion of M.

determinant(M) : real produces the determinant of M.

memlog

memlog(f:&output) : integer writes a message to file f recording the current memory usage in the string and block regions. For each, three figures are written: amount in use, amount reserved, and number of collections. memlog() does not perturb the figures: it requires no allocation itself. memlog() returns the total current usage.

morse
APPENDIX B. THE ICON PROGRAM LIBRARY

morse(s:string) : string converts s to its Morse code equivalent. The version used is known both as International Morse Code and as Continental Code, and is used by radio amateurs (hams).

\[\text{morse}(s:\text{string}) : \text{string} \]

\[\text{converts } s \text{ to its Morse code equivalent.} \]

\[\text{The version used is known both as International Morse Code and as Continental Code, and is used by radio amateurs (hams).} \]

mset

\[
\begin{align*}
\text{same\_value}(d1,d2) : ? & \quad \text{compares d1 and d2 for structural equivalence.} \\
\text{insert2}(S:\text{set}, el) : \text{set} & \quad \text{inserts el into S} \\
\text{member2}(S:\text{set}, el) : ? & \quad \text{tests whether el (or its structural equivalent) is in S.} \\
\text{delete2}(S:\text{set}, el) : \text{set} & \quad \text{deletes el (or its structural equivalent) from S.}
\end{align*}
\]

The mset module implements set operations in which no two identical (structurally equivalent) values can be present in a set.

namepfx

\[
\begin{align*}
\text{namepfx}(s:\text{string}) : \text{string} & \quad \text{produces the "name prefix" from a name in standard form -- omitting any title, but picking up the first name and any initials.} \\
\text{namepfx}() & \quad \text{only knows how to omit common titles found in module titleset.} \\
\text{Namepfx}() & \quad \text{Obviously, it can't always produce the "correct" result.} \\
\text{Links: lastname, titleset.}
\end{align*}
\]

ngrams

\[
\begin{align*}
\text{ngrams(file,n,c:}\&\text{letters},t) : \text{string}^* & \quad \text{generates a tabulation of the n-grams in the specified file.} \\
\text{If c is non-null, it is used as the set of characters from which n-grams are taken (other characters break n-grams).} \\
\text{If t is non-null, the tabulation is given in order of frequency; otherwise in alphabetical order of n-grams.} \\
\text{Note: This procedure is unsuitable if there are very many different n-grams.}
\end{align*}
\]

numbers

\[
\begin{align*}
\text{These procedures format numbers in various ways:} \\
\text{amean}(L) : \text{real} & \quad \text{returns arithmetic mean of numbers in L.} \\
\text{ceil}(r) : \text{integer} & \quad \text{returns nearest integer to r away from 0.} \\
\text{commas}(s) : \text{string} & \quad \text{inserts commas in s to separate digits into groups of three.} \\
\text{decipos}(r;i:3,j:5) : \text{string}? & \quad \text{position decimal point at i in r in field width j.} \\
\text{digred}(i) : \text{integer} & \quad \text{reduces a number by adding digits until one digit is reached.} \\
\text{div}(i:\text{number},j:\text{number}) : \text{real} & \quad \text{produces the result of real division of i by j.} \\
\text{fix}(i,j;w:8,d:3) : \text{string}? & \quad \text{formats } i/j \text{ as a real (floating-point) number in a} \\
\text{field of width w with d digits to the right of the decimal point, if possible.} \\
\text{If w is less than 3 it is set to 3. If d is less than 1, it is set to 1.} \\
\text{The function fails if j is 0 or if the number cannot be formatted.} \\
\text{floor}(r) : \text{integer} & \quad \text{nearest integer to r toward 0.} \\
\text{gcd}(i,j) : \text{integer}? & \quad \text{returns greatest common divisor of i and j. It fails if both are 0.} \\
\text{gcd}(L:\text{list}) : \text{integer} & \quad \text{returns the greatest common division of a list of integers.} \\
\text{gmean}(\text{args}?) : \text{real}? & \quad \text{returns the geometric mean of numbers.} \\
\text{hmean}(\text{args}?) : \text{real}? & \quad \text{returns the harmonic mean of numbers.} \\
\text{large}(i) : \text{integer}? & \quad \text{succeeds if i is a large integer but fails otherwise.}
\end{align*}
\]

(RUN, RJA, RLG, TK)
lcm(i, j) : integer? returns the least common multiple of i and j.
lcm(L): integer? returns the least common multiple of the integers in the list L.
npalins(n) : string* generates palindromic n-digit numbers
roman(i) : string? converts i to Roman numerals.
round(r:real) : integer returns nearest integer to r.
sign(r) : integer returns sign of r.
spell(i : integer) : string? spells out i in English.
trunc(r:real) : integer returns nearest integer less than r.
unroman(string) : integer converts Roman numerals to integers.

Options:

openchk

OpenCheck() causes subsequent opens and closes to write diagnostic information to &errout. Useful for diagnosing situations where many files are opened and closed and there is a possibility that some files are not always being closed.

options

options(args:list, opt:string, err:stop) : table separates, interprets, and removes UNIX-style command options from the argument list of a main() procedure, returning a table of option values. On the command line, options are introduced by a "-" character. An option name is either a single printable character, as in "-n" or "-?", or a string of letters, as in "-geometry". Valueless single-character options may be combined, for example as "-qtv". Some options require values. Generally, the option name is one argument and the value appears as the next argument, for example

"-F file.txt"

However, with a single-character argument name (as in that example), the value may be concatenated: "-F file.txt" is accepted as equivalent. Options may be freely interspersed with non-option arguments. An argument of "-" is treated as a non-option. The special argument "--" terminates option processing. Non-option arguments are returned in the original argument list for interpretation by the caller. An argument of the form @filename (a "@" immediately followed by a file name) causes options() to replace that argument with arguments retrieved from the file "filename". Each line of the file is taken as a separate argument, exactly as it appears in the file.

The options string is a concatenation, with optional spaces between, of one or more option specs of the form -name% where - introduces the option.

name is either a string of letters or any single printable character

% is one of the following flag characters:
! No value is required or allowed
: A string value is required
+ An integer value is required
. A real value is required
The leading "-" may be omitted for a single-character option. The "!" flag may be omitted except when needed to terminate a multi-character name. Thus, the following optstrings are equivalent: "-n+ -t -v -q -F: -geometry: -silent" "n+tvqF:-geometry:-silent" "-silent!n+tvqF:-geometry". If the options string is omitted, any single letter is assumed to be valid and require no data.

The err procedure will be called if an error is detected in the command line options. The procedure is called with one argument: a string describing the error that occurred. After err() is called, options() immediately returns the outcome of errproc(), without processing further arguments. Already processed arguments will have been removed from args.

outbits

outbits(i:integer, len:integer) string* fits variable and/or non-byte-sized blocks into standard 8-bit bytes. outbits() will suspend byte-sized chunks of i converted to characters (most significant bits first) until there is not enough left of i to fill up an 8-bit character. The remaining portion is stored in a buffer until outbits() is called again, at which point the buffer is combined with the new i and then output in the same manner as before. The buffer is flushed by calling outbits() with a null i argument. Note that len gives the number of bits there are in i (or at least the number of bits you want preserved; those that are discarded are the most significant ones). A trivial example of how outbits() might be used:

outtext := open("some.file.name","w")
L := [1,2,3,4]
every writes(outtext, outbits(L,3))
writes(outtext, outbits(&null,3)) # flush buffer

List L may be reconstructed with inbits():

intext := open("some.file.name")
L := []
while put(L, inbits(intext, 3))

Note that outbits() is a generator, while inbits() is not. See also: inbits.icn.

packunpk

pack(num:i,width:i):string produces a binary-coded decimal representation of num
in which each character contains two decimal digits stored in four bits each.
unpack(val:s,width:i):string converts a binary-coded decimal back to a
string representation width characters long of the original source integer.
unpack2(val:string) : integer converts a binary-coded decimal back into its
original source integer.
Links: convert.

partit

partit(i,min:i,max:i):L* generates the partitions of i; that is the ways that i can be
B.1. PROCEDURE LIBRARY MODULES

represented as a sum of positive integers with minimum and maximum values.

\[
\text{partcount}(i, \text{min}:i, \text{max}:i) : \text{integer}
\]
returns the number of partitions.

\[
\text{fibpart}(i) : \text{L}
\]
returns a list of Fibonacci numbers that is a partition of \(i\).

Links: fastfnecs, numbers.

patterns

This module provides string scanning procedure equivalents for most SNOBOL4 patterns and some extensions. Procedures and their pattern equivalents are:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Equivalent SNOBOL</th>
<th>Procedure</th>
<th>Equivalent SNOBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any(s)</td>
<td>\text{ANY}(S)</td>
<td>Arb()</td>
<td>\text{ARB}</td>
</tr>
<tr>
<td>Arbno(p)</td>
<td>\text{ARBNO}(P)</td>
<td>Arb(x)(i)</td>
<td>\text{ARB}(I)</td>
</tr>
<tr>
<td>Bal()</td>
<td>\text{BAL}</td>
<td>Break(s)</td>
<td>\text{BREAK}(S)</td>
</tr>
<tr>
<td>Breakx(s)</td>
<td>\text{BREAKX}(S)</td>
<td>Cat(p1,p2)</td>
<td>P1 P2</td>
</tr>
<tr>
<td>Discard(p)</td>
<td>/P</td>
<td>Exog(s)</td>
<td>\text{	extbackslash}S</td>
</tr>
<tr>
<td>Find(s)</td>
<td>\text{FINDF}(S)</td>
<td>Len(i)</td>
<td>\text{LEN}(I)</td>
</tr>
<tr>
<td>Limit(p,i)</td>
<td>P \text{ \backslash} I</td>
<td>Locate(p)</td>
<td>LOCATE(P)</td>
</tr>
<tr>
<td>Marb()</td>
<td>longest-first \text{ARB}</td>
<td>Notany(p)</td>
<td>\text{NOTANY}(S)</td>
</tr>
<tr>
<td>Pos(i)</td>
<td>\text{POS}(I)</td>
<td>Replace(p,s)</td>
<td>P = S</td>
</tr>
<tr>
<td>Rpos(i)</td>
<td>\text{RPOS}(I)</td>
<td>Rtab(i)</td>
<td>\text{RTAB}(I)</td>
</tr>
<tr>
<td>Span(s)</td>
<td>\text{SPAN}(S)</td>
<td>String(s)</td>
<td>S</td>
</tr>
<tr>
<td>Succeed()</td>
<td>\text{SUCCEED}</td>
<td>Tab(i)</td>
<td>\text{TAB}(I)</td>
</tr>
<tr>
<td>Xform(f,p)</td>
<td>F(P)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following procedures relate to the application and control of pattern matching:

\[
\text{Apply}(s,p) \quad S \ ? P
\]

\[
\text{Mode()} \quad \text{anchored or unanchored matching (see \text{Anchor()} \text{ and } \text{Float()})}
\]

\[
\text{Anchor()} \quad \& \text{ANCHOR} = 1 \text{ if Mode := Anchor}
\]

\[
\text{Float()} \quad \& \text{ANCHOR} = 0 \text{ if Mode := Float}
\]

In addition to the procedures above, the following expressions can be used:

\[
p1() \ | \ p2() \quad \text{P1} \ | \ \text{P2}
\]

\[
v \leftarrow p() \quad \text{P} \ . \ V \ (\text{approximate})
\]

\[
v := p() \quad \text{P} \ . \ V \ (\text{approximate})
\]

\[
\text{fail} \quad \text{FAIL}
\]

\[
= s \quad \text{S} \ (\text{in place of String(s)})
\]

\[
p1() \ || \ p2() \quad \text{P1 P2} \ (\text{in place of Cat(p1,p2)})
\]

Using this system, most SNOBOL4 patterns can be satisfactorily transliterated into Icon procedures and expressions. For example, the pattern

\[
\text{SPAN("0123456789") . N "H" LEN(*N) . LIT}
\]

can be transliterated into

\[
(n \leftarrow \text{Span('0123456789'))} \ || ="H" || (\text{lit} \leftarrow \text{Len}(n))
\]
Concatenation of components is necessary to preserve the pattern-matching properties of SNOBOL4.

**patword**

\[ \text{patword}(s:\text{string}) : \text{string} \]

returns a letter pattern in which each different character in \( s \) is assigned a letter. For example, \( \text{patword}("\text{structural}") \) returns "abcdebdcfg".

**phoname**

\[ \text{phoname}(\text{telno}:\text{string}) : \text{string}^* \]

generates the letter combinations corresponding to the digits in a telephone number. The number of possibilities is very large. This procedure should be used in a context that limits or filters its output.

**plural**

\[ \text{plural}(\text{word}:\text{string}) : \text{string} \]

produces the plural form of a singular English noun. The procedure here is rudimentary and is not correct in all cases.

**polystuf**

These procedures are for creating and performing operations on single-variable polynomials (like \( ax^2 + bx + c \)). A polynomial is represented as a table in which the keys are exponents and the values are coefficients. (EE)

\[ \text{poly}(c_1, e_1, c_2, e_2, ...) : \text{poly} \]

creates a polynomial from the parameters given as coefficient-exponent pairs: \( c_1x^{e_1} + c_2x^{e_2} + ... \)

\[ \text{is}_\text{zero}(n) : ? \]

determines if \( n = 0 \)

\[ \text{is}_\text{zero}_\text{poly}(p) : ? \]

determines if a given polynomial is \( 0x^0 \)

\[ \text{poly}_\text{add}(p_1, p_2) : \text{poly} \]

returns the sum of two polynomials

\[ \text{poly}_\text{sub}(p_1, p_2) : \text{poly} \]

returns the difference of \( p_1 - p_2 \)

\[ \text{poly}_\text{mul}(p_1, p_2) : \text{poly} \]

returns the product of two polynomials

\[ \text{poly}_\text{eval}(p, x) : \text{poly} \]

finds the value of polynomial \( p \) evaluated at the given \( x \).

\[ \text{term}_\text{2string} (c, e) : \text{string} \]

converts one coefficient-exponent pair into a string.

\[ \text{poly}_\text{string}(p) : \text{string} \]

returns the string representation of an entire polynomial.

**printcol**

\[ \text{printcol}(\text{items}, \text{fields}, \text{title}:"", \text{pagelength}:30000, \text{linelength}:80, \text{auxdata}) \]

deals with the problem of printing tabular data where the total width of items to be printed is wider than the page. Simply allowing the data to wrap to additional lines often produces marginally readable output. This procedure facilitates printing such groups of data as vertical columns down the page length, instead of as horizontal rows across the page. That way many, many fields can be printed neatly. The programming of such a transformation can be a nuisance. This procedure does much of the work for you, like deciding how many items can fit across the page width and ensuring that entire items will be printed on the same page without page breaks (if that service is requested).

For example, suppose you have a list of records to print. The record is defined as:

\[ \text{record rec(item1,item2,item3,...)} \]

Also suppose that lines such as
Field 1  Field 2  Field 3 ...
------- ------- ------- ---
Record 1  item1 item2 item3 ...
Record 2  item1 item2 item3 ...

are too long to print across the page. This procedure will print them as:

**TITLE**
=====
Record 1  Record 2 ...
------- ------- ---
Field 1  item1  ...
Field 2  item2  ...
Field 3  item3  ...

The arguments are:

- **items**: a co-expression that produces a sequence of items (usually structured data objects, but not necessarily) for which data is to be printed.
- **fields**: a list of procedures to produce the field’s data. Each procedure takes two arguments. The procedure’s action depends upon what is passed in the first argument:
  - **header** produces the row heading string to be used for that field (the field name).
  - **width** produces the maximum field width (including the column header).
  - **other** produces the field value string for the item passed as the argument.
- **auxdata**: arbitrary auxiliary data to be passed to the field procedures -- see “fields”, above.

---

**printf**

This module provides Icon versions of the `printf()` family of formatted output functions from the C language.

- `printf(fmt:string, args[])` formats and writes arguments to &output.
- `fprintf(f:file,fmt:string,args[])` formats and writes arguments to f.
- `sprintf(fmt:string, args[]) : string` formats arguments and produces a string result.

These procedures support normal d, s, o, and x formats. An "r" format prints real numbers in a manner similar to that of printf’s "f", but will produce a result in an exponential format if the number is larger than the largest integer plus one. Left or right justification and field width control are provided as in printf(). %s and %r handle precision specifications.

---

**prockind**

- `prockind(p:procedure) : string?` produces a code for the kind of the procedure p as follows: "p" (declared) procedure "f" (built-in) function, "o" operator, "c" record constructor. It fails if p is not of type procedure.
Appendix B. The Icon Program Library

**procname**

`procname(p:procedure, x) : string?` produces the name of a procedure (including functions, operators, and record constructors) value. If `x` is null, the result is derived from `image()` in a relatively straightforward way. In the case of operators, the number of arguments is appended to the operator symbol. If `x` is nonnull, the result is put in a form that resembles an Icon expression. `procname()` fails if `p` is not of type procedure.

**pscript**

`epsheader(f, x, y, w, h, flags)` writes an Encapsulated PostScript file header and initializes the PostScript coordinate system. This file contains a procedure for writing PostScript output explicitly, as contrasted with the procedures in `psrecord.icn` that write PostScript as a side effect of normal graphics calls. An EPS file can either be incorporated as part of a larger document or sent directly to a PostScript printer. `epsheader()` writes the first portion of the PostScript output to file `f`. The calling program then generates the rest. It is the caller’s responsibility to ensure that the rest of the file conforms to the requirements for EPS files as documented in the PostScript Reference Manual, second edition. `(x,y,w,h)` specify the range of coordinates that are to be used in the generated PostScript code. `epsheader()` generates PostScript commands that center this region on the page and clip anything outside it. If the flags string contains the letter "r" and `abs(w) > abs(h)`, the coordinate system is rotated to place the region in "landscape" mode. The generated header also defines an "inch" operator that can be used for absolute measurements as shown in the example below.

```icon
f := open(filename, "w") | stop("can't open ",filename)
epsheader(f, x, y, w, h)
write(f, ",07 inch setlinewidth")
write(f,x1, ", y1, " moveto ", x2, ", y2, " lineto stroke") ... write(f, "showpage")
```

**random**

This file contains procedures related to pseudo-random numbers.

`rand_num() : integer` is a linear congruential pseudo-random number generator. Each time it is called, it produces another number in the sequence and also assigns it to the global variable `random`. With no arguments, `rand_num()` produces the same sequence(s) as Icon’s built-in random-number generator. Arguments can be used to get different sequences. The global variable `random` serves the same role that `&random` does for Icon’s built-in random number generator.

`rand_int(i) : integer` produces a random integer in the range 1 to `i`.

`randomize()` sets `&random`, based on the date and time of day.

`randrange(min, max) : integer` produces a random number from `min <= i <= max`.

`randrangeseq(i, j) : integer*` generates the integers from `i` to `j` in random order.

`randseq(seed) : integer*` generates the values of `&random`, starting at `seed`. 
that occur as the result of using \(?x\).

shuffle(x):x shuffles the elements of string, list, or record x. If x is a list or record
it is altered in place instead of creating a new structure for the shuffled result.

Links: factors.

rational
These procedures perform arithmetic on rational numbers (fractions):

record rational(numer, denom, sign) is used to represent rational values.

str2rat(string):rational? converts a string such as "3/2" to a rational number.

rat2str(r:rational):string converts rational number r to its string representation.

addrat(r1:rational,r2:rational) : rational adds rational numbers r1 and r2.

subrat(r1:rational,r2:rational) : rational subtracts rational numbers r1 and r2.

mpyrat(r1:rational,r2:rational) : rational multiplies rational numbers r1 and r2.

divrat(r1:rational,r2:rational) : rational divides rational number r1 by r2.

negrat(r) : rational produces the negative of rational number r.

reciprat(r:rational) : rational produces the reciprocal of rational number r.

Links: numbers.

readtbl (RLG)
readtbl(f:file) : table reads SGML mapping information from a file. This module is part
of the strpsgml package. The file specifies how each SGML tag in a given input text should
be translated. Each line has the form:

SGML-designator start_code end_code

where the SGML designator is something like "quote" (without the quotation marks),
and the start and end codes are the way in which you want the beginning and end of a
<quote>...</quote> sequence to be translated. Presumably, in this instance, your codes
would indicate some set level of indentation, and perhaps a font change. If you don’t have
an end code for a particular SGML designator, just leave it blank. Links: stripunb.

rec2tab
rec2tab(x) : null writes fields of a record as tab-separated string. Carriage returns in
files are converted to vertical tabs. (Works for lists too.)

records
field(R, i) : string? returns the name of the ith field of R. Other record processing
procedures may be added to this module in future editions.

recurmap
recurmap(recur:list) : string maps a recurrence declaration of the form

f(i):
if expr11 then expr12
if expr21 then expr22
...
else expr
The declaration is a list of strings. The result string is a declaration for an Icon procedure that computes corresponding values. At present there is no error checking and the most naive form of code is generated.

---

**reduce**

reduce(op, init, args[]) applies the binary operation op to all the values in args, using init as the initial value. For example, reduce("+", 1, args[]) produces the sum of the values in args.

---

**regexp**

This module implements UNIX-like regular expression patterns. These procedures are interesting partly because of the "recursive suspension" (or "suspensive recursion"): technique used to simulate conjunction of an arbitrary number of computed expressions. String scanning function default conventions are followed.

ReMatch(pattern, s, i1, i2) : integer* produces the sequence of positions in s past a sub-string starting at i1 that matches pattern, but fails if there is no such position. Similar to match(), but is capable of generating multiple positions.

ReFind(pattern, s, i1, i2) : integer* produces the sequence of positions in s where sub-strings begin that match pattern, but fails if there is no such position. Similar to find(). Each position is produced only once, even if several possible matches are possible at that position. pattern can be either a string or a pattern list -- see RePat().

RePat(s) : list? creates a pattern element list from pattern string s, but fails if the pattern string is not syntactically correct. ReMatch() and ReFind() will automatically convert a pattern string to a pattern list, but it is faster to do the conversion explicitly if multiple operations are done using the same pattern. An additional advantage to compiling the pattern separately is avoiding ambiguity of failure caused by an incorrect pattern and failure to match a correct pattern.

ReCaseIndependent() : null, ReCaseDependent() : null set the mode for case-independent or case-dependent matching. The initial mode is case-dependent.

---

**Accessible Global Variables**

After a match, the strings matched by parenthesized regular expressions are left in list Re_ParenGroups, and can be accessed by subscripting it using the same number as the \N construct. If it is desired that regular expression format be similar to UNIX filename generation patterns but still retain the power of full regular expressions, make the following assignments prior to compiling the pattern string:

Re_ArbString := "**" # Defaults to ".**

The csets that define a word, digits, and white space can be modified. The following assignments can be made before compiling the pattern string. The character sets are captured when the pattern is compiled, so changing them after pattern compilation will not alter the behavior of matches unless the pattern string is recompiled.
Re_WordChars := 'whatever' # default: &letters++&digits++'_'
Re_Digits := &digits ++ 'ABCDEFabcdef' # Defaults to &digits
Re_Space := 'whatever' # Defaults to ' 	\v\n\r\f'

These globals are initialized in the first call to RePat() if they are null. They can be explicitly initialized to their defaults (if they are null) by calling Re_Default().

Characters compiled into patterns can be passed through a user-supplied filter procedure, provided in global variable Re_Filter. Filtering is done as the pattern is compiled, before the characters are bound into the pattern. The filter is passed one argument, the string to filter, and it must return the filtered string as its result. If the filter procedure fails, the string will be used unfiltered. The filter is called with an argument of either type string (for characters in the pattern) or cset (for character classes [...]). By default, individual pattern elements are matched in a "leftmost-longest-first" sequence, which is the order observed by Perl, egrep, and most other regular expression matchers. The following global variable setting causes the matcher to operate in leftmost-shortest-first order.

Re_LeftmostShortest := 1

In the case of patterns containing alternation, ReFind() will generally not produce positions in increasing order, but will produce all positions from the first term of the alternation (in increasing order) followed by all positions from the second (in increasing order). If it is necessary that the positions be generated in strictly increasing order, with no duplicates, assign a non-null value to Re_Ordered:

Re_Ordered := 1

The Re_Ordered option imposes a small performance penalty in some cases, and the co-expression facility is required in your Icon implementation.

Regular Expression Characters and Features Supported  The regular expression format supported by procedures in this file model very closely those supported by the UNIX "egrep" program, with modifications as described in the Perl programming language definition. Following is a brief description of the special characters used in regular expressions. The abbreviation RE means regular expression.

c  An ordinary character (not one of the special characters discussed below) is a one-character RE that matches that character.

\c  A backslash followed by any special character is a one-character RE that matches the special character itself. Note that backslash escape sequences representing non-graphic characters are not supported directly by these procedures. Of course, strings coded in an Icon program will have such escapes handled by the Icon translator. If such escapes must be supported in strings read from the run-time environment (e.g. files), they will have to be converted by other means, such as the Icon Program Library procedure escape().


A period is a one-character RE that matches any character.

[string] A non-empty string enclosed in square brackets is a one-character RE that matches any one character in that string. If the first character is “ˆ” (circumflex), the RE matches any character not in the remaining characters of the string. The “-” (minus), when between two other characters, may be used to indicate a range of consecutive ASCII characters (e.g. [0-9] is equivalent to [0123456789]). Other special characters stand for themselves in a bracketed string.

* Matches zero or more occurrences of the RE to its left.
+ Matches one or more occurrences of the RE to its left.
? Matches zero or one occurrences of the RE to its left.
{N} Matches exactly N occurrences of the RE to its left.
{N,} Matches at least N occurrences of the RE to its left.
{N,M} Matches from N to M occurrences of the RE to its left.
* A caret at the beginning of an entire RE constrains that RE to match an initial substring of the subject string.
$ A currency symbol at the end of an entire RE constrains that RE to match a final substring of the subject string.
| Alternation: two regular expressions separated by ”|” match either a match for the first or a match for the second.
() A RE enclosed in parentheses matches a match for the regular expression (parenthesized groups are used for grouping, and for accessing the matched string subsequently in the match using the \N expression).
\N where N is a digit in the range 1-9, matches a string of characters that was matched by a parenthesized RE to the left in the same RE. The sub-expression specified is that beginning with the Nth occurrence of "(" counting from the left. e.g., ^(.*)\1. matches a string consisting of two consecutive occurrences of the same string.

Extensions beyond UNIX egrep The following extensions to UNIX regular expressions, as specified in the Perl programming language, are supported.

\w matches any alphanumeric (including ".").
\W Matches any non-alphanumeric.
\b matches at a word-boundary (a word is a string of alphanumerics, as in \w).
\B matches only non-word-boundaries.
\s matches any white-space character.
\S matches any non-white-space character.
\d matches any digit [0-9].
\D matches any non-digit.
\w, \W, \s, \S, \d, \D can be used within [string] regular expressions.
repetit(L: list) : integer returns the length of the smallest range of values that repeat in a list. For example, if \( L := [1, 2, 3, 1, 2, 3, 1, 2, 3] \), \( \text{repetit}(L) \) returns 3. If there is no repetition, \( \text{repetit()} \) returns the length of the list.

rewrap

rewrap(s: string, i: 70) : string? reformats text fed to it into strings \(<i\) in length. \( \text{rewrap}() \) utilizes a static buffer, so it can be called repeatedly with different string arguments, and still produce homogeneous output. This buffer is flushed by calling \( \text{rewrap}() \) with a null first argument. Here's a simple example of how \( \text{rewrap}() \) could be used. The following program reads the standard input, producing fully rewrapped output.

```plaintext
procedure main()
    every write(rewrap(!&input))
    write(rewrap())
end
```

Naturally, in practice you would want to do things like check for indentation or blank lines in order to wrap only on a paragraph by paragraph basis, as in

```plaintext
procedure main()
    while line := read(&input) do {
        if line == "" then {
            write("" \(\Rightarrow\) \(\text{rewrap}()\))
            write(line)
        } else {
            if match("\t", line) then {
                write(rewrap())
                write(rewrap(line))
            } else
                write(rewrap(line))
        }
    }
end
```

Fill-prefixes can be implemented simply by prepending them to the output of rewrap:

```plaintext
i := 70; fill_prefix := " > "
while line := read(input_file) do {
    line ?:= (f_bit := tab(many('> ')) | "", tab(0))
    write(fill_prefix || f_bit || rewrap(line, i - *fill_prefix))
    etc.
```
Obviously, these examples are fairly simplistic. Putting them to actual use would certainly require a few environment-specific modifications and/or extensions. Still, they offer some indication of the kinds of applications rewrap() might be used in. Note: If you want leading and trailing tabs removed, map them to spaces first. rewrap() only fools with spaces, leaving tabs intact. This can be changed easily enough, by running its input through the Icon detab() function. See also: wrap.icn.

This module contains procedures related to string scanning: Where indicated by parameter names, they follow the string scanning conventions for parameter defaults.

\[
\text{balq}(c1, c2, c3, c4: `\"\", c5: `\', s, i1, i2): integer*
\]
generates integer positions in \( s \) preceding a character of \( c1 \) in \( s[i1:i2] \) that are (a) balanced with respect to characters in \( c2 \) and \( c3 \) and (b) not "quoted" by characters in \( c4 \) with "escape" sequences in \( c5 \).

\[
\text{balqc}(c1, c2, c3, c4: `\"\", c5, s1: `/*`, s2: `*/`, s3, i1, i2): integer*
\]
is like \( \text{balq}() \) with the addition that balanced characters within "comments", as delimited by the strings \( s1 \) and \( s2 \), are also excluded from balancing. In addition, if \( s1 \) is given and \( s2 \) is null then the comment terminates at the end of string.

\[
\text{limatch}(L: list, c: cset): integer?
\]
matches items in \( L \) delimited by characters in \( c \). Returns the last cursor position scanned to, or fails

\[
\text{slashbal}(c1, c2, c3, s, i, j): integer*
\]
behaves like \( \text{bal}() \), except that it ignores, for purposes of balancing, any \( c2 \) or \( c3 \) char which is preceded by a backslash.

\[
\text{slashupto}(c, s, i, j): integer?
\]
works just like \( \text{upto}() \), except that it ignores backslash escaped characters.

\[
\text{snapshot}(title: string, len: integer)
\]
writes a snapshot of the state of string scanning, showing the value of \&subject \ and \&pos \ Two optional arguments specify a title written at the top of the snapshot, and a width (in characters) at which to wrap output. For example,

\[
"((a+b)-delta)/(c*d))" \ ? \ \{tab(bal(\'+\-\')); snapshot("example")\}
\]
produces

\[
---example---------------------------
|                                |
|                                |
|                                |
| &subject = "((a+b)-delta)/(c*d))" |
|                                |
|                                |

Note that the bar showing \&pos \ is positioned under the \&pos\textsuperscript{th} character (actual positions are between characters). If \&pos \ is at the end of \&subject, the bar is positioned under the quotation mark delimiting the subject. Escape sequences are handled properly.
scanset

scan_setup(s, i1, i2) : scan_setup_result? sets things up for user-written string-scanning procedures that are in the spirit of Icon’s built-ins. The values passed are the last three arguments to all Icon scanning functions (such as upto(c,s,i1,i2)). scan_setup() supplies any appropriate defaults and returns needed values. The value returned is a record scan_setup_result(ss, offset) where ss is the substring of s to be scanned, and offset is the size of the substring of s that precedes the substring to be scanned. scan_setup() fails if i1 or i2 is out of range with respect to s. The user-written procedure can then match in the string ss to compute the position within ss appropriate to the scan (p). The value returned (or suspended) to the caller is p + offset (the position within the original string, s). For example, the following function finds two words separated by spaces:

procedure two_words(s,i1,i2)
  local x,p
  x := scan_setup(s,i1,i2) | fail # fail if out of range
  x.ss ? suspend {
    tab(upto(&letters)) &
    pos(1) | (move(-1) & tab(any(~&letters))) &
    p := &pos & # remember starting position
    tab(many(&letters)) & tab(many(" ")) &
    tab(many(&letters)) &
    p + x.offset # return position in original s
  }
end

segment

These procedures segment a string s into consecutive substrings consisting of characters that respectively do/do not occur in c. (WHM)

segment(s,c) : string* generates the substrings, while seglist(s,c) : list produces a list of the segments. For example, segment("Not a sentence.",&letters) generates six string results: "Not" " " "a" " "sentence" "." while seglist("Not a sentence.",&letters) produces a list of size six: ["Not"," ","a","sentence","."]

sentence, senten1

Two alternative modules provide a function sentence(f) :string* that generates sentences from file f. A lot of grammatical and stylistic analysis programs are predicated on the notion of a sentence. For instance, some programs count the number of words in each sentence. Others count the number and length of clauses. Still others pedantically check for sentence-final particles and prepositions. Neither module’s definition of a sentence will handle all possible inputs properly; you may wish to try both of them to see which one works better on your inputs.

Module sentence requires co-expressions, while module senten1 does not. Module senten1 uses a definition of a "sentence" detailed in the source code.
**seqimage**

*Seqimage(e,i,j): string* produces a string image of the result sequence for the expression *e*. The first *i* results are printed. If *i* is omitted, there is no limit. If there are more than *i* results for *e*, ellipses are provided in the image after the first *i*. If *j* is specified, at most *j* results from the end of the sequence are printed after the ellipses. If *j* is omitted, only the first *i* results are produced. For example, the expressions

```
Seqimage{1 to 12}
Seqimage{1 to 12,10}
Seqimage{1 to 12,6,3}
```

produce, respectively,

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10,...}
{1, 2, 3, 4, 5, 6, ..., 10, 11, 12}
```

If *j* is given and *e* has an infinite result sequence, *Seqimage{}* does not terminate.

**sername**

*sername(p:"file", s:"", n:3, i:0) : string* produces a series of names of the form *p<nnn>*s. If *n* is given it determines the number of digits in <nnn>. If *i* is given it resets the sequence to start with *i*. <nnn> is a right-adjusted integer padded with zeros. Ordinarily, the arguments only are given on the first call. Subsequent calls without arguments give the next name. For example, *sername("image", ".gif", 3, 0)* produces "image000.gif", and subsequently, *sername()* produces "image001.gif", "image002.gif", and so on. If any argument changes on subsequent calls, all non-null arguments are reset.

**sets**

*sets (AB, REG)*

```
cset2set(c:cset): set returns a set that contains the individual characters in *c*.
domain(T:table) returns the domain (set of keys) of the function defined by *T*.
inverse(T:table, x) : table returns the inverse of the function defined by *T*. If *x* is null, it’s the functional inverse. If *x* is an empty list, it’s the relational inverse. If *x* is an empty set, it the relational inverse, but with each table member as a set instead of a list.
pairset(T:table) : set converts *T* to an equivalent set of ordered pairs.
range(T:table):set returns the range (set of values) of the function defined by *T*.
seteq(S1, S2) : set? tests equivalence of sets *S1* and *S2*.
setlt(S1, S2) : set? tests inclusion (strict subset) of set *S1* in *S2*.
```

**showtbl**

*showtbl(title:"", T, mode, limit, order, posit, w1:10, w2:10, gutter:3, f1:left, f2:right)* displays table *T* according to the arguments given. The remaining arguments are:
B.1. PROCEDURE LIBRARY MODULES

mode indicates the type of sorting, one of: "ref"|"val" (by key or decreasing value)
limit specifies the maximum lines of table output, if any
order gives the sort order, one of: "incr"|"decr" (not implemented yet)
posit is the first column position, one of: "ref"|"val" (not implemented yet)
w1 is the width of 1st column
w2 is the width of 2nd column
gutter is the width between columns
f1 supplies the formatting function used on the 1st column
f2 supplies the formatting function used on the 2nd column
showtbl() returns a record with the first element being a count of the size of the table
and the second element the number of lines written.

shquote (RJA)

This module is useful for writing Icon programs that generate shell commands. Certain
characters cannot appear in the open in strings that are to be interpreted as "words" by
command shells. This family of procedures assists in quoting such strings so that they will
be interpreted as single words. Quoting characters are applied only if necessary -- if strings
need no quoting they are returned unchanged.

shquote(s1, s2,..., sN) : string produces a string of words s1, s2,..., sN that are
properly separated and quoted for the Bourne Shell (sh).
cshquote(s1, s2,..., sN) : string produces a string of words s1, s2,..., sN that are
properly separated and quoted for the C-Shell (csh).
mpwquote(s1, s2,..., sN) : string produces a string of words s1, s2,..., sN that are
properly separated and quoted for the Macintosh Programmer’s Workshop shell.
dequote(s1,s2:"\") : string produces the UNIX-style command line word s1 with any
quoting characters removed. s2 is the escape character required by the shell.

signed (RJA)

signed(s) : integer puts raw bits of characters of string s into an integer. The value is
taken as signed. This procedure is normally used for processing of binary data read from
a file.

sort (RJA,RLG,REG)

sortff(L, fields[]) is like sortf(), except it takes an unlimited number of field arguments.
sortgen(T, m) : any generates sorted output in a manner given by m:
"k+" sort by key in ascending order
"k-" sort by key in descending order
"v+" sort by value in ascending order
"v-" sort by value in descending order
sortt(T, i) is like sort(T, i) but produces a list of two-element records instead of a list of
two-element lists.

soundex, soundex1 (CW, JDS)

statemap
statemap() : table produces a "two-way" table to map state names (in the postal sense) to their postal abbreviations and vice-versa. The list is done in two parts with auxiliary procedures so that this procedure can be used with the default constant-table size for the translator and linker.

str2toks
Scanning procedure str2toks(c:"(letters++digits), s, i1, i2) : string* suspends portions of s[i1:i2] delimited by characters in c. str2toks() is not a primitive scanning function in the sense that it suspends strings, and not integer positions. The code:

"hello, how are ya?" ? every write(str2toks())

writes to &output, on successive lines, the words "hello", "how", "are", and finally "ya" (skipping the punctuation). The beginning and end of the line count as delimiters. Note that if i > 1 or j < *s+1 some tokens may end up appearing truncated.

strings
These procedures perform operations on strings.
cat(s1, s2,...) : string concatenates an arbitrary number of strings.
charcnt(s, c) : integer returns the number of instances of characters in c in s.
collate(s1, s2) : string collates the characters of s1 and s2. For example, collate("abc", "def") produces "adbecf".
comb(s, i) : string* generates combinations of characters of s taken i at a time.
compress(s, c:&cset) : string compresses consecutive occurrences of members of c in s.
csort(s) : string produces the characters of s in lexical order.
decollate(s, i:1) : string produces a string consisting of every other character of s. If i is odd, the odd-numbered characters are selected, while if i is even, the even-numbered characters are selected.
deletec(s, c) : string deletes occurrences of characters in c from s.
deletep(s, L) : string deletes all characters at positions specified in L.
deletes(s1, s2) : string deletes occurrences of s2 in s1.
diffcnt(s) : integer returns count of the number of different characters in s.
extend(s, n) : string replicates s to length n.
interleave(s1, s2) : string interleave characters s2 extended to the length of s1 with s1. ispal(s) : string? succeeds and returns s if s is a palindrome.
maxlen(L, p:proc("*", 1)) : integer returns the length of the longest string in L. p is applied to each string as a "length" procedure.
meander(s, n) : string produces a "meandering" string that contains all n-tuples of characters of s.

minlen(L, p: proc("**", 1)) : integer returns the length of the shortest string in L. p is applied to each string as a "length" procedure.

ochars(s) : string produces unique characters in the order that they appear in s.

palins(s, n) : string* generates the n-character palindromes from characters in s.

permute(s) : string* generates all the permutations of the string s.

pretrim(s, c:' ') : string trims characters from beginning of s.

reflect(s1, i, s2:"") : string returns s1 concatenated with s2 and the reversal of s1 to produce a partial palindrome. The values of i determine "end conditions" for the reversal: 0 = omit first and last characters, 1 = omit first character, 2 = omit last character, 3 = don’t omit character.

replace(s1, s2, s3) : string replaces all occurrences of s2 in s1 by s3.

replacem(s,...) : string performs multiple replacements in the style of replace(), where multiple argument pairs may be given, as in replacem(s, "a", "bc", "d", "cd") which replaces all a’s by "bc" and all d’s by "cd". Replacements are performed sequentially, not in parallel.

replc(s, L) : string replicates characters of s by amounts given by the values in L.

rotate(s, i:1) : string rotates s i characters left (negative i rotates to the right).

schars(s) : string produces the unique characters of s in lexical order.

scramble(s) : string scrambles (shuffles) the characters of s randomly.

selectp(s, L) : string selects characters of s that are at positions given in L.

transpose(s1, s2, s3) : string transposes s1 using label s2 and transposition s3.

stripcom

stripcom(s) : string? strips the commented-out portion of a line of code. Fails on lines which, either stripped or otherwise, come out as an empty string. stripcom() can’t handle lines ending in an underscore as part of a broken string literal, since stripcom() is not intended to be used on sequentially read files. It simply removes comments from individual lines.

stripunb

stripunb(c1,c2,s,i,j,t:table) : string strips material from a line which is unbalanced with respect to the characters defined in arguments 1 and 2 (unbalanced being defined as bal() defines it, except that characters preceded by a backslash are counted as regular characters, and are not taken into account by the balancing algorithm). If you call stripunb() with a table argument as follows, stripunb('<','>',s,&null,&null,t) and if t is a table having the form

key: "bold" value: outstr("\e[2m","\e1m")
key: "underline" value: outstr("\e[4m","\e[1m") etc.

then every instance of "<bold>" in string s will be mapped to "\e2m," and every instance of "</bold>" will be mapped to "\e[1m." Values in table t must be records of type outstr(on,
When "</>" is encountered, stripunb() will output the .off value for the preceding .on string encountered. Links: scan.

tab2list, tab2rec

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tab2list(string):list</td>
<td>takes tab-separated strings and inserts them into a list. Vertical tabs in strings are converted to carriage returns.</td>
</tr>
<tr>
<td>tab2rec(s, x): null</td>
<td>takes tab-separated strings and assigns them into fields of a record or list x. Vertical tabs in strings are converted to carriage returns. See also: list2tab.icn, rec2tab.icn.</td>
</tr>
</tbody>
</table>

For the operations on tables that mimic set operations, the correspondences are only approximate and do not have the mathematical properties of the corresponding operations on sets. For example, table "union" is not symmetric or transitive. Where there is potential asymmetry, the procedures "favor" their first argument. The procedures that return tables return new tables and do not modify their arguments.

tclass

tclass(x): string | returns "atomic" or "composite" depending on the type of x. |

title, titleset

title(name:string): string | produces the title of a name, such as "Mr."
"Mr. John Doe". The process is imperfect. Links titleset. |
titleset(): set | produces a set of strings that commonly appear as titles in names. This set is (necessarily) incomplete. |

trees

depth(t): integer | compute maximum depth of tree t |
| ldag(s): list | construct a DAG from the string s |
| ltree(s): list | construct a tree from the string s |
| stree(t): string | construct a string from the tree t |
tcopy(t): list | deep copy tree t. |
teq(t1,t2): list? | compare trees t1 and t2 |
visit(t) : any* visit, in preorder, the nodes of the tree t by suspending them all

This module provides tree operations using a list representation of trees and directed acyclic graphs (DAGs). The procedures do not protect themselves from cycles.

tuple

tuple(tl: list) : list implements a "tuple" feature that produces the effect of multiple keys. A tuple is created by an expression of the form tuple([expr1, expr2, ..., exprn]). The result can be used in a case expression or as a table subscript. Lookup is successful provided the values of expr1, expr2, ..., exprn are the same (even if the lists containing them are not). For example, consider selecting an operation based on the types of two operands. The following expression uses tuple() to drive a case expression using value pairs.

```
case tuple([type(op1), type(op2)]) of {
    tuple(["integer", "integer"]): op1 + op2
    tuple(["string", "integer"]): op1 || "+" || op2
    tuple(["integer", "string"]): op1 || "+" || op2
    tuple(["string", "string"]): op1 || "+" || op2
}
```

typecode

typecode(x) : string produces a one-letter string identifying the type of its argument. In most cases, the code is the first (lowercase) letter of the type, as in "i" for the integer type. Structure types are in uppercase, as in "L" for the list type. All records have the code "R". The code "C" is used for the co-expression type to avoid conflict for the "c" for the cset type. The code "w" is produced for windows.

unsigned

unsigned(s) : integer puts raw bits of characters of string s into an integer. The value is taken as unsigned. This procedure is normally used for processing of binary data read from a file.

usage

These procedures provide various common services:
Usage(s) stops execution with a message indicating how to use a program.
Error(args?) writes arguments to &errout on a single line, preceded by "*** ".
ErrorCheck(l,f) : null reports to &output an error that was converted to failure.
Feature(s) : ? succeeds if feature s is available in this implementation.
Requires(s) terminates execution if feature s is not available.
Signature() writes the version, host, and features support in the running implementation of Icon.

varsub

varsub(s, varProc:getenv) obtains a variable value from the procedure, varProc. As with the UNIX Bourne shell and C shell, variable names are preceded by $. Optionally,
the variable name can additionally be surrounded by curly braces {}, which is usually done when necessary to isolate the variable name from surrounding text. As with the C-shell, the special symbol `~<username>` is handled. Username can be omitted; in which case the value of the variable HOME is substituted. If username is supplied, the `/etc/passwd` file is searched to supply the home directory of username (this action is obviously not portable to non-UNIX environments). (RJA)

```plaintext
version
    version() : string? produces the version number of Icon on which a program is running. It only works if the &version is in the standard form.
```

vrml, vrml1lib, vrml2lib

These modules contain procedures for producing VRML files.

- `point_field(L)`: create VRML point field from point list L
- `u_crd_idx(i)`: create VRML coordinate index for 0 through i - 1
- `render(x)`: render node x
- `vrml1(x)`: produces VRML 1.0 file for node x
- `vrml2(x)`: produces VRML 2.0 file for node x
- `vrml_color(s)`: converts Icon color specification to vrml form

Notes: Not all node types have been tested. Where field values are complex, as in vectors, these must be built separately as strings to go in the appropriate fields. There is no error checking. Fields must be given in the order they appear in the node record declarations and field values must be of the correct type and form. The introduction of record types other than for nodes will cause bogus output. A structural loop will produce output until the evaluation stack overflows.

- `vrml1lib.icn` contains record declarations for VRML 1.0 nodes. Note: Although VRML 1.0 allows node fields to be given in any order, they must be specified in the order given in the record declarations that follow. Omitted (null-valued) fields are ignored on output. Group nodes require list arguments for lists of nodes.

- `vrml2lib.icn` contains record declarations for VRML 2.0 nodes. Note: Although VRML 2.0 allows node fields to be given in any order, they must be specified in the order given in the record declarations that follow. Group nodes require list arguments for lists of nodes.


```plaintext
wdiag
    wdiag(s1, s2,...) : null writes the values of global variables s1, s2, ... with s1, s2, ... as identifying labels. It writes a diagnostic message to standard error if an argument is not the name of a global variable.
```

```plaintext
weighted
    WeightedShuffle(sample, percentage) : list returns the list sample with a portion of the elements switched. Examples:
    WeightedShuffle(X, 100) - returns a fully shuffled list
```
B.1. PROCEDURE LIBRARY MODULES

**WeightedShuffle(X, 50)** - every other element is eligible to be switched

**WeightedShuffle(X, 25)** - every fourth element is shuffled

**WeightedShuffle(X, 0)** - nothing is changed

The procedure will fail if the given percentage is not between 0 and 100, inclusive, or if it is not a numeric value.

---

**wildcard**

These procedures deal with UNIX-like filename wild-card patterns containing *, ?, and [...], as found in the UNIX shells csh and sh; they are described briefly in the **wild_pat()** procedure. Recursive suspension is used to simulate conjunction of an arbitrary number of computed expressions. Default values of s, i1, and i2 are the same as for Icon’s built-in string scanning procedures such as **match()**. The public procedures are:

**wild_match(pattern,s,i1,i2) : integer** produces the sequence of positions in s past a substring starting at i1 that matches pattern, but fails if there is no such position. Similar to **match()**, but is capable of generating multiple positions.

**wild_find(pattern,s,i1,i2) : integer** produces the sequence of positions in s where substrings begin that match pattern, but fails if there is no such position. Similar to **find()**. pattern can be either a string or a pattern list, see **wild_pat()**, below.

**wild_pat(s) : L** creates a pattern element list from pattern string s. A pattern element is needed by **wild_match()** and **wild_find()**. **wild_match()** and **wild_find()** will automatically convert a pattern string to a pattern list, but it is faster to do the conversion explicitly if multiple operations are done using the same pattern.

---

**word**

String scanning function **word(s, i1, i2) : integer** produces the position past a UNIX-style command line word, including quoted and escaped characters.

**word_dequote(s) : string** produces the UNIX-style command line word s with any quoting characters removed. Links: **scanset**.

---

**wrap**

**wrap(s:"",i:0) : string** facilitates accumulation of small strings into longer output strings, outputting when the accumulated string would exceed a specified length (e.g. outputting items in multiple columns). s is the string to accumulate, i is the width of desired output string. **wrap()** fails if the string s did not necessitate output of the buffered output string; otherwise the output string is returned (which never includes s). Calling **wrap()** with no arguments produces the buffer (if it is not empty) and clears it. **wrap()** does no output to files. Here’s how **wrap()** is normally used:

```
wrap() # Initialize; not needed unless there was a previous use.
every i := 1 to 100 do # Loop to process strings to output --
    write(wrap(x[i],80)) # only writes when 80-char line filled.
write(wrap()) # Output what’s in buffer, if something to write.
```
APPENDIX B. THE ICON PROGRAM LIBRARY

wraps(s,i) : string? is similar to wrap(), but intended for use with writes(). If the string s did not necessitate a line-wrap, s is returned. If a line-wrap is needed, s, preceded by a new-line character ("\n"), is returned.

xcodes (RJA, REG)

This module provides procedures to save and restore Icon structures to disk. Module xcodes handles the encoding of records using canonical names: record0, record1, ... This allows programs to decode files by providing declarations for these names when the original declarations are not available. xcodes also provides for procedures and files present in the encoded file that are not in the decoding program.

xencode(x,f) stores x in file f such that it can be converted back to x by xdecode(f). The procedures handle most kinds of values, including structures of arbitrary complexity and even loops. The following sequence will output x and recreate it as y:

```
f := open("xstore","w")
xencode(x,f)
close(f)
f := open("xstore")
y := xdecode(f)
close(f)
```

For "scalar" types null, integer, real, cset, and string, this sequence will result in the relationship x === y. For structured types (list, set, table, record and object) y is not identical to x, but it has the same "shape" and its elements bear the same relation to the original as if they were encoded and decoded individually. Files, co-expressions, and windows are not handled; they are restored as empty lists so that (1) they will be unique objects and (2) will likely generate run-time errors if they are (probably erroneously) used in computation. However, the special files &input, &output, and &errout are restored. Not much can be done with functions and procedures, except to preserve type and identification. The encoding of strings and csets handles all characters in a way that it is safe to write the encoding to a file and read it back. xdecode() fails if given a file that is not in xcode format or if the encoded file contains a record for which there is no declaration in the program in which the decoding is done. Of course, if a record is declared differently in the encoding and decoding programs, the decoding may be bogus.

xencoden() and xdecoden() perform the same operations, except they take the name of a file to open, not an already-open file.

xencodet() and xdecodet() are like xencode() and xdecode() except that the trailing argument is a type name. If the encoded decoded value is not of that type, they fail. xencodet() does not take an opt argument.

xencode(x, f:&output, p:write) : f encodes x writing to file f using procedure p that writes a line on f using the same interface as write() (the first parameter is always the value passed as f).
xencode(f:&input, p:read) : x returns the restored object where f is the file to read, and p is a procedure that reads a line from f using the same interface as read() (the parameter is always the value passed as f). The p parameter is not normally used for storage in text files, but it provides the flexibility to store the data in other ways, such as a string in memory. If p is provided, then f can be any arbitrary data object -- it need not be a file. For example, to "write" x to an Icon string:

```icon
record StringFile(s)
procedure main() ...
encodeString := xencode(x,StringFile(""),WriteString).s
...
end
procedure WriteString(f,s[])
every f.s ||:= !s
f.s ||:= "\n"
return
end
```

Links: escape. See also: object.icn, codeobj.icn.

---

xforms

These procedures produce matrices for affine transformation in two dimensions and transform point lists. A point list is a list of Point() records.

- transform(p:list, M) : list transforms a point list by matrix
- transform_points(pl:list,M) transforms point list
- set_scale(x, y) : matrix produces a matrix for scaling
- set_trans(x, y) : matrix produces a matrix for translation
- set_xshear(x) : matrix produces a matrix for x shear
- set_yshear(y) : matrix produces a matrix for y shear
- set_rotate(x) : matrix produces a matrix for rotation

Links: gobject. See also: matrix. (SBW, REG)

---

ximage

ximage(x) : s produces a string image of x. ximage() differs from image() in that it outputs all elements of structured data types. The output resembles Icon code and is thus familiar to Icon programmers. Additionally, it indents successive structural levels in such a way that it is easy to visualize the data’s structure. Note that the additional arguments in the ximage() procedure declaration are used for passing data among recursive levels.

- xdump(x1,x2,...,xn) : xn uses ximage() to successively write the images of x1, x2, ..., xn to &errout. For example, the code

```icon
t := table() ; t["one"] := 1 ; t["two"] := 2
xdump("A table",t)
xdump("A list",[3,1,3,[2,4,6],3,4,3,5])
```
writes the following output (note that \texttt{ximage()} infers the predominant list element value and avoids excessive output):

\begin{verbatim}
"A table"
T18 := table(&null)
T18["one"] := 1
T18["two"] := 2
"A list"
L25 := list(8,3)
L25[2] := 1
L24[1] := 2
L24[2] := 4
L25[8] := 5
\end{verbatim}

\texttt{xrotate}

\texttt{xrotate(X, i)} rotates the values in \texttt{X} right by one position. It works for lists and records.

This procedure is mainly interesting as a recursive version of

\begin{verbatim}
x1 :=: x2 :=: x3 :=: ... xn
\end{verbatim}

since a better method for lists is \texttt{push(L, pull(L))}.

\section*{B.2 Application Programs, Examples, and Tools}

The Icon Program Library \texttt{progs} directory contains 200+ programs that are useful for demonstration, entertainment, and/or practical utility. When it is available, usage information is presented with [optional] and repeated* command-line arguments.

\begin{itemize}
  \item \texttt{adlcheck, adlcound, adlfilter, adlfirst, adllist, adlsort}
  \item \texttt{adlcheck} checks address lists for correctness.
  \item \texttt{adlcound} counts all the entries in an address list file. If an argument is given, it counts only those that have designators with characters in the argument.
  \item \texttt{adlfilter} filters address lists, allowing through only those entries with specified selectors. The option \texttt{-s arg} selects entries with characters in args (default is all). Option \texttt{-x} inverts the logic, selecting characters not in args
  \item \texttt{adlfirst} writes all the first lines of entries in an address list file. If an argument is given, it counts only those that have designators with characters in the argument.
  \item \texttt{adllist} lists entries in address lists.
  \item \texttt{adlsort} sorts entries in address lists.
\end{itemize}

Read the source for a description of these programs’ many command line options.

See also: address.doc, labels.icn.
animal (RJA)

*animal* is the familiar "animal game". The program asks the user a series of questions in an attempt to guess what animal he or she is thinking of. This expert system gets smarter as it plays. Typing "list" at any yes/no prompt will show an inventory of animals known, and there are some other commands too.

banner (CT)

*banner* is a utility that outputs enlarged letters (5x6 matrix) in portrait mode.

bj (CT, RLG)

*bj* is a simple but fun blackjack game. The original version was for an ANSI screen. This version has been modified to work with the UNIX termcap database file.

blnk2tab

*blnk2tab* is a UNIX-style filter that converts strings of two or more blanks to tabs.

c2icn (RJA)

The *c2icn* filter does easy work involved in porting a C program to Icon. It reformats comments, moves embedded comments to end of line, removes ";" from ends of lines, reformats line-continued strings, changes = to :, and changes -> to .

calc

*calc* is a simple Polish "desk calculator". It accepts as values integers, reals, csets, and strings, as well as an empty line for the null value. Other lines of input are operations, either binary operators, functions, or one of: clear (remove all values from the calculator’s stack), dump (write out the contents of the stack), or quit (exits from the calculator). Failure and most errors are detected; arguments are consumed and not left on the stack.

chkhtml (RJA)

This program checks an HTML file and detects the following errors: Reference to undefined anchor name; duplicated anchor name; warning for unreferenced anchor name; unknown tag; badly formed tag; improper tag nesting; unescaped <, >, " or &; bad escape string; improper embedding of attributes; and bad (non-ASCII) characters. The program also advises on the use of *<HTML>*, *<HEAD>*, and *<BODY>* tags.

colm [-w linewidth] [-s space] [-m min_width] [-t tab_width] [-x] [-d] file*

*colm* arranges a number of data items, one per line, into multiple columns. Items are arranged in column-wise order, that is, the sequence runs down the first column, then down the second, etc. If a null line appears in the input stream, it signifies a break in the list, and the following line is taken as a title for the following data items. No title precedes the initial sequence of items. See the program source code for command line options interpretation.

comfiles

*comfiles* lists common file names in two directories given as command-line arguments. Requires: UNIX
**APPENDIX B. THE ICON PROGRAM LIBRARY**

concord

**concord** produces a simple concordance from standard input to standard output. Words less than three characters long are ignored. Option `-l n` sets maximum line length to `n` (default 72), after which **concord** starts a new line. Option `-w n` sets the maximum width for word to `n` (default 15), after which words are truncated.

conman (WED)

**conman** responds to queries like "? Volume of the earth in tbsp". The commands (which are not reserved) are: **load**, **save**, **print**, `?` (same as print), **list**, **is**, **are** (same as is).

countlist

**countlist** counts how many times each line of input occurs and writes a summary. By default, the output is sorted first by decreasing count and within each count, alphabetically. Option `-a` causes output to be sorted alphabetically; `-t` prints a total at the end.

cross (WPM)

**cross** takes a list of words and tries to arrange them in crossword format so that they intersect. Uppercase letters are mapped into lowercase letters on input. The program objects if the input contains a nonalphabetic character. It produces only one possible intersection and it does not attempt to produce the most compact result.

crypt [key] <infile >outfile (PB, PLT)

**crypt** is an example encryption program. Do not use this in the face of competent cryptanalysis. Helen Gaines’ book (Gaines, 1939) is a classical introduction to the field.

csgen

**csgen** accepts a context-sensitive production grammar and generates randomly selected sentences from the corresponding language. Uppercase letters stand for nonterminal symbols and -> indicates the left-hand side can be rewritten by the right-hand side. Other characters are considered to be terminal symbols. Lines beginning with `#` are comments and are ignored. A line consisting of a nonterminal symbol followed by a colon and a non-negative integer `i` is a generation specification for `i` instances of sentences for the language defined by the nonterminal (goal) symbol. See the program source code for example inputs and output, additional options, and usage explanation.

cstrings (RJA)

**cstrings** prints all strings (enclosed in double quotes) in C source files.

daystil (NL)

**daystil** calculates the number of days between the current date and the date given on the command line, which may be specified in many ways. For example, August 12 can be specified as "August 12", "Aug 12", "12 August", or "12 aUGuS", among others.

deal
deal shuffles, deals, and displays hands in the game of bridge. Option -h n produces n hands. The default is 1.

declchck

declchck examines ucode files and reports declared identifiers that may conflict with function names. Requires: UNIX.

delamc (TRH)

delamc delaminates standard input into several output files according to the separator characters specified by the string following the -t option (default: tab). All output files contain the same number of lines as the input file.

detex (CLJ)

detex reads in documents written in the LaTeX typesetting language, and removes some of the common LaTeX commands to produce plain ASCII. Output must typically be further edited by hand to produce an acceptable result.

diffn file*

diffn shows the differences between n files.

diffsum [file] (GMT)

diffsum reads output from the Unix diff(1) utility, either normal diffs or context diffs. For each pair of files compared, diffsum reports two numbers: The number of lines added or changed, and the net change in file size. The first of these indicates the magnitude of the changes and the second the net effect on file size.

diffu f1 f2 (RM)

diffu uses dif() to generate file differences like the UNIX diff(1) command.

diffword

diffword lists the different words in the input. The definition of "word" is naive.

duplfile

duplfile lists the file names that occur in more than one subdirectory and the subdirectories in which the names occur. This program runs slow on large directory structures.

envelope [options] < address(es) (RF)

envelope addresses envelopes on a PostScript or HP-LJ printer, including barcodes for the zip code. A line beginning with # or an optional alternate separator can be used to separate multiple addresses. The parser will strip the formatting commands from an address in a troff or LaTeX letter. Typically, envelope is used from inside an editor. In Emacs, mark the region of the address and do M-| envelope. In vi, put the cursor on the first line of the address and do :,:N w !envelope where N = number-of-lines-in-address.

farb, farb2

Dave Farber, co-author of the original SNOBOL programming language, is noted for his creative use of the English language. Hence the terms "farberisms" and "to farberate".
This program produces a randomly selected farberism.

filecnvt [-i s1] [-o s2] infile outfile  

filecnvt copies a text file, converting line terminators. Option -i s1 indicates the system for which the input file has line termination (default "u"). Option -o s2 directs output to use line terminators for system s2 (default "u"). The designations are: d for DOS/Windows ("\n\r"), m for Macintosh ("\r"), and u UNIX ("\n").

fileprnt

fileprnt reads the file specified as a command-line argument and writes out a representation of each character in several forms: hex, octal, decimal, symbolic, and ASCII.

filesect

filesect start nlines writes the section of the input file starting at a line number and extending some number of lines. If the specifications are out of range, the file is truncated.

filtskel  

filtskel is a skeleton/template for creation of filter programs. Command line options, file names, and tabbing are handled; you need only provide the filtering code.

findstr

findstr is a utility filter to list character strings embedded in data files such as object files. The option -l length gives the minimum string size to be printed (default 3); option -c chars specifies a string of characters (besides the standard ASCII printable characters) to be considered part of a string. String escape sequences can be used.

findtext

findtext retrieves multiline text from a database indexed by idxtxt. Each section of text lines follows a line declaring the index terms. Each index line begins with "::".

fixpath filename oldpath newpath

fixpath changes file paths or other strings in a binary file by modifying the file in place. Each null-terminated occurrence of oldpath is replaced by newpath. If the new path is longer than the old one, a warning is given and the old path is extended by null characters, which must be matched in the file for replacement to take place. This is dangerous in general but allows repairing an errant fixpath command.

format

format is a filter program that word-wraps a range of text. Options include full justification. All lines that have the same indentation as the first line (or same comment leading character format if -c option) are wrapped. Other lines are left as is. This program is useful in conjunction with editors that can invoke filters on a range of selected text.

former

former takes a single line of input and outputs it in lines no greater than the number given on the command line (default 80).
fract (REG, GMT)

fract produces successive rational approximations to a real number. The option -n r specifies the real number to be approximated, default .6180339887498948482. Option -l i gives the limit on number of approximations, default 100. This program was based on a calculator algorithm posted by: Joseph D. Rudmin, Duke Univ. Physics Dept., 1987.

fuzz (AC)

fuzz does "fuzzy" string pattern matching. The result of matching s and t is a number between 0 and 1, based on counting matching pairs of characters in increasing substrings of s and t. Characters may be weighted differently.

gcomp (WHM, REG)

gcomp produces a list of the files in the current directory that do not appear among the arguments. For example, gcomp *.c produces a list of files in the current directory that do not end in .c. Requires: UNIX.

genqueen (PAB)

genqueen n solves the non-attacking n-queens problem for (square) boards of arbitrary size. The problem consists of placing chess queens on an n-by-n grid such that no queen is in the same row, column, or diagonal as any other queen. The output is each of the solution boards; rotations not considered equal.

gftrace (GMT)

gftrace writes a set of procedures to standard output. Those procedures can be linked with an Icon program to enable the tracing of calls to built-in functions. See the comments in the generated code for details. The set of generated functions reflects the built-in functions of the version of Icon under which this generator is run.

graphdem (MH)

graphdem is a simple bar graphics package with two demonstration applications. The first displays the 4 most frequently used characters in a string; the second displays the Fibonacci numbers. Requires: ANSI terminal support.

groups (TRH)

groups sorts input containing "records" defined to be groups of consecutive lines. One or more repetitions of a demarcation line beginning with the separator string separate each input record. The first line of each record is used as the key. The separator string defaults to the empty string; empty lines are default demarcation lines.

headicon

headicon prepends a standard header to an Icon program. It does not check to see if the program already has a header. The first command-line argument is taken as the base name of the file; default "foo". The second command-line argument is taken as the author; the default is "Ralph E. Griswold" — but you can personalize it for your own use. The new
file is brought up in the vi editor. The file skeleton.icn must be accessible via `dopen()`.

---

### hebcalen, hcal4unx

(ADC, RLG)

The Jewish year harmonizes the solar and lunar cycle, using the 19-year cycle of Meton (c. 432 BCE). Startup syntax is `hebcalen [date]`, where `date` is a year specification of the form 5750 for a Jewish year, +1990 or 1990AD or 1990CE or -1990 or 1990BC or 1990BCE for a civil year. Requires: keyboard functions, `hebcalen.dat`, `hebcalen.hlp`.

---

### hr

(CT)

hr implements a horse-race game.

---

### ibar

ibar replaces comment bars by bars 76 characters long, the IPL standard.

---

### ibrow

(RJA)

ibrow browses Icon files for declarations, defaulting to browse all *.icn files in the current directory. The user interface is self-explanatory -- use "??" for help if you’re confused.

---

### icalc

(SBW)

icalc is an infix calculator with control structures and compound statements. It illustrates a technique that can greatly reduce the performance cost of recursive-descent parsing with backtracking. Features include: integer and real value arithmetic; variables; function calls to built-in functions; strings allowed as function arguments; unary operators + (absolute value) and - (negation); assignment :=; binary operators: +, -, *, /, %, and ^; relational operators: =, !=, <, <=, >, >= (use 1 for true and 0 for false); compound statements in curly braces with semicolon separators; if-then and if-then-else; while-do; and limited multiline input. The input is processed one line at a time, in calculator fashion, but compound statements can be continued across line boundaries.

---

### icalls

icalls processes trace output and tabulates calls of procedures.

---

### icn2c

(RJA)

icn2c partially converts Icon to C. It reformats comments, line-continued strings and procedure declarations, changes `:=` to `=`, and changes `end` to `}`.

---

### icontent <options> <Icon source file>...

(RJA)

icontent builds a list, in Icon comment format, of procedures and records in an Icon source file. Multiple files can be specified as arguments, and are processed in sequence. Option `-s` specifies to sort names alphabetically (default is in order of occurrence); option `-l` lists in single column (default is to list in multiple columns).

---

### icvt

(CW, REG)

icvt converts Icon programs from ASCII syntax to EBCDIC syntax or vice versa. The option `-a` converts to ASCII, while the option `-e` converts to EBCDIC. The program given in standard input is written in converted form to standard output.
B.2. APPLICATION PROGRAMS, EXAMPLES, AND TOOLS  

idepth  

idepth processes trace output and reports the maximum depth of recursion.

idxtext [-a] file1 [file2 [...]] (RLG, PLT)  

idxtext turns a file associated with the gettext() routine into an indexed text-base. Though gettext() will work fine with files that haven’t been indexed via idxtex(), access is faster if the indexing is done if the file is, say, over 5k or 10k. file1, file2, etc. are the names of gettext-format files that are to be (re-)indexed. The -a flag tells idxtex() to abort if an index file already exists. Indexed files have the format:

keyname delimiter offset [delimiter offset [etc.]]

ifilter op  

ifilter applies the operation given on the command-line to each line of standard input, writing the results. For example, ifilter reverse <foo writes the lines of foo reversed end-for-end, while ifilter right 10 0 <foo calls right(line, "10", "0") for each line of foo.

igrep (RJA)  

igrep emulates UNIX egrep using the enhanced regular expressions supported by regexp.icn. Options are nearly identical to those supported by egrep (no -b: print disk block number). The option -E allows Icon-type (hence C-type) string escape sequences in the pattern string. Beware: when -E is used, backslashes that are meant to be processed in the regular expression context must be doubled.

iheader  

iheader lists the headers of Icon files whose names are given on the command line.

ihelp [-f helpfile] [item] [keyword ...] (RJA)  

ihelp displays help information. The optional item name specifies the section of the help file that is to be displayed. If no item name is specified a default section will be displayed, which usually lists the help items that are available.

iidecode [infile] [-x], iiencode [infile] [-x] remotefile [-o outfile] (RLG, FJL)  

These are Icon ports of the UNIX/C uuencode/uuencode utilities, based on freely distributable BSD code. File modes are always encoded with 0644 permissions. An option specifies xxencoded files (like uuencoding, but passes unscathed through EBCDIC sites).

ilnkxref [-options] <icon source file>... (RJA)  

ilnkxref is a utility to create cross-reference of library files used in Icon programs (i.e., those files named in link declarations). Requires: UNIX. Option -p specifies to sort by "popularity"; option -v tells the program to report progress information.

ilump  

ilump copies Icon source files, incorporating recursively the source code for files named by link directives. This produces a whole source program in one file, useful with certain
profiling and visualization tools. If a linked file is not found in the current directory, directories specified by the LPATH environment variable are tried.

**imagetyp**

*imagetyp* accepts file names from standard input and writes their image type to standard output. It relies on a procedure *imagetyp(s)* that attempts to determine the type of image file named *s*. Corrupted or fake files easily fool it. Examples of some image file types were not available for testing. The types presently recognized are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Image file type</th>
<th>Value</th>
<th>Image file type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xpm</td>
<td>XPM image</td>
<td>ps</td>
<td>PostScript document</td>
</tr>
<tr>
<td>pgm</td>
<td>PGM image</td>
<td>cgm char</td>
<td>Computer Graphics Metafil, character</td>
</tr>
<tr>
<td>?</td>
<td>unknown</td>
<td>cgm text</td>
<td>Computer Graphics Metafil, text</td>
</tr>
<tr>
<td>iris</td>
<td>Iris image</td>
<td>sundraw</td>
<td>SunDraw document</td>
</tr>
<tr>
<td>pbm</td>
<td>PBM image</td>
<td>cgm binary</td>
<td>Computer Graphics Metafil, binary</td>
</tr>
<tr>
<td>ppm</td>
<td>PPM image</td>
<td>xwd</td>
<td>X Window dump</td>
</tr>
<tr>
<td>bmp</td>
<td>BMP image</td>
<td>gif</td>
<td>Compuserv GIF image</td>
</tr>
<tr>
<td>xmp</td>
<td>XMP image</td>
<td>ras</td>
<td>UNIX raster image</td>
</tr>
<tr>
<td>pcx</td>
<td>PCX image</td>
<td>rle</td>
<td>UNIX RLE image</td>
</tr>
<tr>
<td>tiff</td>
<td>TIFF image</td>
<td>iff</td>
<td>IFF/ILBM image</td>
</tr>
</tbody>
</table>

**ineeds** *(RJA)*

*ineeds* determines Icon modules required by an Icon module. It expects environment variable LPATH to be set properly as for the Icon Compiler.

**inter**

*inter* lists lines common to two files.

**interpe, interpp** *(REG, JN)*

*interpe* is a crude but effective interpreter for Icon expressions. Each line entered from standard input must be an Icon expression. If the expression is a generator, all its results are produced. If option -e is given, the expression is echoed.

*interpp* accepts Icon expressions prefixed with line numbers. You can resequence them, list them, and execute the lines in order. Retype a line to change it. There is a "?" help command that lists the other commands.

**ipatch** *(GMT)*

*ipatch* changes the path to iconx, the Icon interpreter, which is embedded in an Icon executable file under Unix. Because the headers of such files are not designed to expand, a different form of header is written to accommodate a possibly longer path.

**ipldoc**

*ipldoc* collects selected information from documentation headers for Icon procedure files named on the command line. Option -s skips file headers; option -f sorts the procedure list by file, instead of the default sort by procedure name.
iplindex [-k i:16] [-p i:12]  

iplindex produces an indexed listing of the Icon Program Library. Option -k i gives the width keyword field; option -p i specifies the width of field for program name. If a file except.wrd is readable in the current directory, the words in it are used instead.

iplkwic  

iplkwic is a specialized version of kwic.icn used for producing kwic listings for the Icon program library. This is a simple keyword-in-context (KWIC) program. The "key" words are aligned at a specified column, with the text shifted as necessary. Text shifted left is truncated at the left. Tabs and other special characters may not be handled properly.

iplweb [-ipl source] [dest]  

iplweb generates web pages from IPL header comments. Environment variable IPL locates the Icon Program Library if -ipl is not specified; dest defaults to the current directory. iplweb generates an HTML subdirectory in dest and makes an index to gprogs, gprocs, procs, and progs directories under HTML. These directories contain an .html file for each .icn file in the referenced directory. An index to all files is also generated. The .html files contain the IPL standard comment header info inside.

iprint  

iprint formats Icon programs, trying to keep whole procedures on the same page. It identifies the end of a print group (e.g. procedure) by looking for the string (defaulting to "end") at the beginning of a line. Comments and declarations prior to the procedure are grouped with the procedure. Page creases are skipped over, and form-feeds (ˆL) embedded in the file are handled properly. Page headings (file name, date, time, page number) are printed unless suppressed by the -h option. See the program source code for details.

ipsort  

ipsort reads an Icon program and writes an equivalent program with the procedures reordered. Global, link, and record declarations come first and aren’t reordered. The main() procedure comes next followed by the remaining procedures in alphabetical order.

ipsplit  

ipsplit reads an Icon program and writes each procedure to a separate file. The output file names consist of the procedure name with .icn appended. If -g is specified, any global, link, and record declarations are written to that file. Otherwise they are written in the file for the procedure that immediately follows them.

ipxref  

ipxref cross-references Icon programs. It lists the occurrences of each variable by line number. Variables are listed by procedure or separately as globals. The options specify the formatting of the output and whether or not to cross-reference quoted strings and non-alphanumeric. Variables that are followed by a left parenthesis are listed with an asterisk
following the name. If a file is not specified, then standard input is cross-referenced. The following options change the defaults:

- `-c n` The column width (default: 4) per line number.
- `-l n` The left margin or starting column (default: 40) of the line numbers.
- `-w n` The column width (default: 80) of the whole output line.

Normally only alphanumerics are cross-referenced. Two options broaden it:

- `-q` Include quoted strings.
- `-x` Include all non-alphanumerics.

This program assumes the subject file is a valid Icon program. For example, it expects quotes be matched. Bugs: In some situations, the output is not properly formatted.

**isrcline**

`isrcline` counts the number of lines in an Icon program that actually contain code, as opposed to being comments or blank lines.

**istrip**

`istrip` removes comments, empty lines, and leading spaces from an Icon program.

**itab**

`itab` [options] [source-program...]  

`itab` entabs an Icon program, leaving quoted strings alone. The options are:
- `-i` Input tab spacing (default 8)
- `-o` Output tab spacing (default 8).

`itab` observes Icon conventions for escapes and continuations in string constants. If no source-program names are given, standard input is "itabbed" to standard output.

**itags**

`itags` [-aBFtvwx] [-f tagsfile] file...

`itags` creates a tags file for an Icon program. The options are:
- `-a` append output to an existing tags file.
- `-B` use backward searching patterns (??...??).
- `-F` use forward searching patterns (/.../)(default).
- `-x` produce object names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output. This simple index can be printed out as an off-line readable function index.
- `-t` create tags for records.
- `-v` produce on the standard output an index of the form expected by vgrind(1).

This listing contains the function name, file name, and page number (assuming 64 line pages). Since the output will be sorted into lexicographic order, it may be desired to run the output through sort -f. Sample use:

`itags -v files | sort -f > index vgrind -x index`

- `-w` suppress warning diagnostics.

**itrbksum**

`itrbksum` summarizes traceback information produced on error termination by filtering
out the bulk of the procedure traceback information.

**itrcfltr**

*itrcfilter* filters trace output. If there are command-line arguments, they are taken as procedure names, and only those lines with those names are written. If there are no command-line arguments, all lines are written. The names of procedures to pass through can be given in a "response" file as accepted by `options()`, as in

```
itrcfltr @names <trace_file
```

where `names` is a file containing the names to be passed through. The following option is supported: `-a` list all trace messages; overrides any procedure names given.

**itrcsum**

*itrcsum* provides a summary of Icon trace output.

**iundec**

(iundec) invokes `icont` to find undeclared variables in an Icon source program. The output is in the form of a "local" declaration, preceded by a comment line that identifies that procedure and file name from whence it arose. Beware that undeclared variables aren’t necessarily local, so any which are intended to be global must be removed from the generated list. Multiple files can be specified as arguments, and are processed in sequence. The program works only if procedures are formatted such that the keywords "procedure" and "end" are the first words on their respective lines. Requires: pipe support.

**iwriter**

*iwriter* reads `&input` and writes code that when compiled and executed, writes out the original input. This is handy for incorporating message text in programs.

**knapsack**

(knapsack) is a filter that solves a knapsack problem - how to fill a container to capacity by inserting items of various volumes. Its input consists of a string of newline-separated volumes. Its output is a single solution. knapsack demonstrates an underlying algorithm that might be useful in a variety of real-world applications. Knapsack may be tested conveniently by supplying its standard input with any sequence of random numbers.

**krieg**

(Krieg) implements a variation of the game of chess which has the same rules and goal as ordinary chess except that neither player sees the other’s moves or pieces. Kriegspiel combines the intricacies and flavor of chess with elements of uncertainty that characterize games of imperfect information such as bridge or poker.

**kross**

*kross* accepts pairs of strings on successive lines. It diagrams all the intersections of the two strings in a common character.

**kwic, kwicprep**

(kwic) (kwicprep)
kwic is a simple keyword-in-context (KWIC) program. The "key" words are aligned in column 40, with the text shifted as necessary. Text shifted left is truncated at the left. Tabs and other characters whose "print width" is less than one may not be handled properly. If an integer is given on the command line, it overrides the default 40. Some noise words are omitted (see "exceptions" in the program text). If a file named except.wrd is open and readable in the current directory, the words in it are used instead. kwicprep prepares information to create keyword-in-context listings of the Icon program library.

labels

labels produces mailing labels using coded information taken from the input file. In the input file, a line beginning with # is a label header. Subsequent lines up to the next header or end-of-file are accumulated and written out centered horizontally and vertically on label forms. Lines beginning with * are treated as comments and are ignored. See the source code for options. See also: address.doc, adl*.icn, zipsort.icn.

lam files

lam laminates files named on the command line, producing a concatenation of corresponding lines from each file named. If files are different lengths, empty lines are substituted for missing lines in the shorter files. Argument -s causes string s to be inserted between the concatenated file lines. Each command line argument is placed in the output line at the point that it appears in the argument list. For example, lines from file1 and file2 can be laminated with a colon between each line from file1 and the corresponding line from file2 by lam file1 -: file2. File names and strings may appear in any order in the argument list. If a file is named more than once, its lines will be duplicated on the output line, but if standard input is named more than once, its lines will be read alternately. For example, each pair of lines from standard input can be joined onto one line with a space between them by the command lam - "-" - while lam file1 "- " file1 replicates each line from file1.

latexidx processes LaTeX idx files. Input: A latex .idx file containing the \indexentry lines. Output: \item lines sorted in order by entry value, pagerefs in sorted order.

lc simply counts the number of lines in standard input and writes the result to standard output. Assumes UNIX-style line terminators, and uses lots of memory.

lindcode

lindcode reads a file of L-system specifications and build Icon code that creates a table of records containing the specifications. If the option -e is given, symbols for which there is no definition are included in the table with themselves as replacement. See also: lindrec.icn.

lineseq

lineseq reads values on separate lines and strings them together on a single line. The separating strings is specified by the -s option (default: ").
**lisp** (SBW, PLT)

*lisp* is a simple interpreter for pure Lisp. It takes the name of the Lisp program as a command-line argument. The syntax and semantics are based on EV-LISP, as described in L. Siklossy’s "Let’s Talk LISP" (Prentice-Hall, 1976). Functions that have been predefined match those described in Chapters 1-4 of the book. No attempt at improving efficiency has been made; this is rather an example of how a simple Lisp interpreter might be implemented in Icon. The language implemented is case-insensitive. It only reads enough input lines at one time to produce at least one lisp-expression, but continues to read input until a valid lisp-expression is found. Errors: Fails on EOF; fails with error message if current input cannot be made into a valid lisp-expression (i.e. more right than left parentheses).

**literat** (MH)

*literat* is a database system to manage information concerning literature. The program uses files *literat.fil*, *literat2.fil* and *adress.fil* to store its data on the disk. It has a predefined structure of the items and field labels to make it easy to use and to cut down the source code length. Requires: ANSI terminal support.

**loadmap** (SBW)

*loadmap* produces a formatted listing of selected symbol classes from a compiled object file. The listing is by class, and gives the name, starting address, and length of the region associated with each symbol. If the address of a symbol cannot be determined, ???? is given in its place. The size of the last region in a symbol class is suspect and is usually given as rem. Output is not particularly exciting on a stripped file.

**longest**

*longest* writes the (last) longest line in the input file. If the command-line option -# is given, the number of the longest line is written first.

**lower**

*lower* maps the names of all files in the current directory to lowercase.


*makepuzz* takes a list of words, and constructs out of them one of those square find-the-word puzzles that some people like to bend their minds over. *infil* is a file containing words, one to a line, and *outfil* is the file to write the puzzle. *ht* (height) and *wd* (width) are the basic dimensions you want to try to fit your word game into (default 20x20). If the -s argument is present, *makepuzz* will scramble its output.

**missile** (CT)

*missile* is a text-based Missile Command game. It runs on systems that support the delay() function, and uses ANSI escape sequences for screen output. To play use 7, 8, and 9 to launch a missile. 7 is leftward, 8 is straight, and 9 is right. q quits the game.

**miu** (CAC, REG)

**mkpasswd**

`mkpasswd` creates a list of randomly generated passwords consisting of eight random characters in the range [A-Z0-9]. Number of passwords to generate is the first argument.

**monkeys**

The old monkeys at the typewriters anecdote... `monkeys` uses ngram analysis to randomly generate text in the same 'style' as the input text.

**morse**

If `morse` is invoked without arguments, a Morse code table is printed. If arguments are given, the Morse code conversion is printed in dots and dashes. If the first argument’s first character is a dot or dash, the arguments are Morse code and converted to a string.

**mr**

`mr [recipient] [-u user] [-f spool]`

With no arguments, `mr` reads the default mail spool. Another user, a spool file, or the recipient for outgoing mail can be given as a command line argument. Help, including the symbols used to indicate the status of mail, is available with the H command. The program is oriented towards UNIX Internet mail processing.

**newicon**

`newicon` creates a new file with a standard Icon program header and a skeleton mail procedure. The first command-line argument is taken as the base name of the file; default "foo". The second command-line argument is taken as the author. The default is "Ralph E. Griswold"; personalize it for your own use. The same comment applies to the skeleton file mentioned below. The new file is brought up in the vi editor. The options are:

- `-f` overwrite an existing file
- `-p` produce a procedure file instead of a program
- `-o` provide program skeleton with `options()`

The files `skeleton.icn`, `skelproc.icn`, and `skelopt.icn` must be accessible via `dopen()`.

**newsrc**

`newsrc` takes the `.newsrc` file, moves active groups to the beginning, and then appends inactive groups with the numbers omitted, followed by anything else. The groups are alphabetized. The user may retain a set of groups at the top of the file by specifying how many groups on the command line. If not specified, it will be prompted for. The new file is called newnewsrc. The user can replace `.newsrc` with it if it is satisfactory.

**nim**

The game of `nim` is played with a pile of 15 sticks. Each player can select 1, 2, or 3 sticks from the pile on their turn. The player to pick up the last stick(s) wins. The loser of the previous game always gets to go first. There are two versions of nim here. The default...
B.2. APPLICATION PROGRAMS, EXAMPLES, AND TOOLS

version uses an algorithm that will never lose if it gets the first turn. The second version
learns from each game. After a while it will also never lose if it gets the first turn, if you
know how to play. To invoke the learning version, pass an argument to the program. show
displays the program’s game memory after each game. save writes a file called ".nimdump"
in the current directory with a dump of the program’s game memory when you quit. When
the game is played in learn mode it will initialize its game memory from the dump. You
can run this program with both show and save at the same time.

oldicon

oldicon updates the date line in a standard Icon program header. The old file is saved
with the suffix .bak. The file then is brought up in the vi editor unless the -f option is
specified. Requires: system(), vi(1), UNIX.

pack

pack packages a list of files named on the command line into a single file, which is
written to standard output. Files are separated by a header, #######, followed
by the file name. This simple scheme does not work if a file contains such a header itself,
and it’s problematical for files of binary data. See also: unpack.icn.

paginate

paginate processes a document text file, inserting form feeds at appropriate places.

papply

papply applies the procedure given as a command-line argument to each line of standard
input, writing out the results. For example, papply reverse <foo
writes out the lines of foo reversed end-for-end. There is no way to provide other arguments. Except for use with
(built-in) functions, this program must be linked with procedures that are used with it.

parens

parens produces parenthesis-balanced strings in which the parentheses are randomly
distributed. See the source code for options. This program was motivated by the need for
test data for error repair schemes for block-structured programming languages. A useful
extension to this program would be a way to generate other text among the parentheses.

pargen

pargen reads a context-free grammar and produces an Icon program that parses the
Corresponding language. Nonterminal symbols are enclosed in angle brackets. ::= separates
the LHS and RHS of the production. Vertical bars separate alternatives. Other characters
are considered to be terminal symbols. The nonterminal on the first line is the goal.

Parentheses can be used for grouping symbols, as in <term>::=<element>(|*<term>)
Note that an empty alternative is allowable. The right-hand side metacharacters \(<, >, (, )\), and | are accessible through the built-in symbols <lb>, <rb>, <lp>, <rp>, and <vb>,
respectively. There are two other build-in symbols, <empty> and <nl> that match the
empty string and a newline, respectively. Lines beginning with a = are passed through
unchanged, allowing Icon declarations to be placed in the parser. Lines beginning with a # are comments and are ignored. If the name of a ucode file is given on the command line, a link declaration for it is provided in the output. Limitations: Left recursion in the grammar may cause the parser to loop. See also: recog.icn, matchlib.icn, and parscond.icn.

parse, parsex

parse parses simple statements. It provides an interesting example of the use of co-expressions. parsex is another expression parser, adapted from C code written by Allen I. Holub (Dr. Dobb’s Journal, Feb 1987). This general-purpose expression analyzer can evaluate any expression consisting of numbers and the following operators (listed according to precedence level):

() ! 'str'str' * / & + - < <= > >= != && ||

All operators associate left to right unless () are present. The top - is a unary minus.

patchu

patchu reads a source file and a diff file, producing an updated file. The diff file may be generated by the UNIX diff(1) utility, or by diffu.icn, which uses dif.icn. The original patch(1) utility, written by Larry Wall, is widely used in the UNIX community. See diff(1) in a UNIX manual for more details. Requires: co-expressions.

pdecomp

pdecomp lists the prime factors of integers given in standard input.

polydemo

polydemo demonstrates the polystuf module. The user can create, output, delete, or operate up to 26 polynomials, indexed by letter. Available commands: (R)ead - allows input of a polynomial by giving pairs of coefficients and exponents. For example, entering 5, 6, 2, and 3 will create 5x^6 + 2x^3. This polynomial will be stored by an index that is a lower-case letter. (W)rite - outputs to the screen a chosen polynomial. (A)dd - adds two polynomials and defines the sum as a third (S)ubtract - subtracts two polynomials and defines the difference as a third. (M)ultiply - multiplies two polynomials and defines the product as a third. (E)valuate - gives the result of setting x in a polynomial to a value (C)lear - deletes one polynomial (H)elp - lists all commands (Q)uit - end the demonstration.

post [-n groups] [-s subj] [-d dist] [-f followup] [-p prefix] [file]

post posts a news article to Usenet. Given the name of a file containing a news article, post creates a follow-up article, with an attribution and quoted text. On systems posting via inews, post validates newsgroups and distributions in the active and distributions files in the news library directory. Newsgroup validation assumes the active file is sorted.

press

press is a file archiving utility that contains extensive tracing facilities that illustrate the LZW compression process in detail. Compression can be turned off if faster archiving is desired. The LZW compression procedures in this program are general purpose and
B.2. APPLICATION PROGRAMS, EXAMPLES, AND TOOLS

suitable for reuse in other programs.

**procprep**

*procprep* is used to produce the data needed to index the "#:" comments on procedure declarations that is needed to produce a permuted index to procedures.

**procwrap**

*procwrap* takes procedure names from standard input and writes minimal procedure declarations for them. For example, the input line wrapper produces:

```plaintext
procedure wrapper()
end
```

This program is useful when you have a lot of procedures to write.

**psrsplit**

*psrsplit* separates *psrecord.icn* output pages. If a file produced by the procedures in *psrecord.icn* contains multiple pages, it cannot be easily incorporated into another document. *psrsplit* reads such a file and breaks it into individual pages. For an input file named *xxxx* or *xxxx.yyy*, the output files are named *xxxx.p01*, *xxxx.p01*, etc. for as many pages as are available. It is assumed that the input file was written by *psrecord.icn*; the likelihood of correctly processing anything else is small.

**puzz**

*puzz* creates word search puzzles.

**qei**

*qei* takes expressions entered at the command line and evaluates them. A semicolon is required to complete an expression. If one is not provided, the subsequent line is added to what already has been entered. It is important to know that *qei* accumulates expressions and evaluates all previously entered expressions before it evaluates a new one. A line beginning with a colon is a command. The commands are: *:clear* clears the accumulated expressions; *:every* generates all the results from the expression, otherwise, at most one is produced; *:exit* or *quit* terminates the session; *:list* lists the accumulated expressions; and *:type* toggles the switch that displays the type of the result (default: on).

**qt**

*qt* writes out the time in English. If -a is present, only the time is printed: *just after a quarter to three*. Otherwise, time is printed as a sentence: *It's just after a quarter to three*.

**queens**

*queens* displays the solutions to the non-attacking n-queens problem: the ways in which n queens can be placed on an n-by-n chessboard so that no queen can attack another. An integer command line argument specifies the number of queens (default: 6). For example, *queens -n8* displays the solutions for 8 queens on an 8-by-8 chessboard. This program is worth reading for its programming techniques.

**ranstars**
ranstars displays a random field of "stars" on an ANSI terminal at randomly chosen positions on the screen until the specified maximum number is reached. It then extinguishes existing stars and creates new ones for the specified steady-state time, after which the stars are extinguished, one by one. The programming technique is worth noting.

**reply** [prefix] < **news-article** or **mail-item**  

*reply* creates the appropriate headers and attribution, quotes a news or mail message, and uses *system()* calls to put the user in an editor and then to mail the reply. The default prefix for quoted text is " > " . If a smarthost is defined, Internet addresses are converted to bang paths (*name@site.domain* becomes *site.domain!name*). The mail is routed to a domained smarthost as *address@smarthost.domain*, otherwise to *smarthost!address*. The default editor can be overridden with the EDITOR environment variable.

**repro**  

*repro* is the shortest known self-reproducing Icon program.

**revsort**  

*revsort* sorts strings with characters in reverse order.

**roffcmds**  

*roffcmds* processes standard input and writes a tabulation of nroff/troff commands and defined strings to standard output. Limitations: the program only recognizes commands that appear at the beginning of lines and does not attempt to unravel conditional constructions. Similarly, defined strings buried in disguised form in definitions are not recognized.

**rsg**  

*rsg* generates randomly selected strings ("sentences") from a grammar. Grammars are context-free extended BNF similar to that used in *pargen*. The program works interactively, allowing the user to build, test, modify, and save grammars. Input to *rsg* consists of various specifications, which can be intermixed: Productions define nonterminal symbols in syntax similar to the rewriting rules of BNF with various alternatives consisting of the concatenation of nonterminal and terminal symbols. Generation specifications cause the generation of a specified number of sentences from the language defined by a given nonterminal symbol. Grammar output specifications cause the definition of a specified nonterminal or the entire current grammar to be written to a given file. Source specifications cause subsequent input to be read from a specified file. Any line beginning with = causes the rest of that line to be used as a prompt to the user whenever *rsg* is ready for input (there normally is no prompt). A line consisting of a single = stops prompting.

Specifying a new production for a nonterminal symbol changes its definition. There are a number of special devices to facilitate the definition of grammars, including: `<lb>` for `<`, `<rb>` for `>`, `<vb>` for `|`, `<nl>` for newline, `<>` for the empty string, `<&lcase>` denoting any single lowercase letter, `<&ucase>` for any single uppercase letter, and `<&digit>` representing any single digit (Note: &digit, not &digits). In addition, if the string between a `<` and a `>` begins and ends with a single quotation mark, it stands for any single character
between the quotation marks. For example, `<xyz>` is equivalent to `x|y|z`

**ruler**

*ruler* writes a character ruler to standard output. The first optional argument is the length of the ruler in characters (default 80). The second is a number of lines to write, with a line number on each line.

**scramble**

*scramble* takes a document and re-outputs it in a cleverly scrambled fashion. It uses the next two most likely words to follow. The concept was found in a recent Scientific American and Icon seemed to offer the best implementation.

**setmerge file [op] file**...

*setmerge* combines sets of items according to the specified operators. Sets are read from files, one entry per line. Operation is from left to right without any precedence rules. After all operations are complete the resulting set is sorted and written to standard output.

**shar text_file**...

*shar* creates a Bourne shell archive of text files.

**shortest**

*shortest* writes the (last) shortest line in the input file. If the command-line option `-#` is given, the number of the shortest line is written first.

**shuffle**

*shuffle* writes a version of the input file with the lines shuffled.

**sing**

*sing* is an Icon adaptation of a SNOBOL program by Mike Shapiro in (Griswold71). The procedure *sing()* writes the lyrics to the song, "The Twelve Days of Christmas" to a parameter that can be any file open for output. It would be especially nice to send the lyrics to a speech synthesizer (perhaps via a pipe). The algorithm used can be adapted to other popular songs, such as "Old McDonald had a Farm".

**snake [character]**

While away the idle moments watching the snake eat blank squares on your screen. *character* represents a single character to be used in drawing the snake. The default is an "o". To run *snake*, your terminal must have cursor movement and able to do reverse video.

**solit**

*solit* was inspired by a solitaire game that was written by Allyn Wade in 1985. This program supports several common terminals and PC's. Note: The command-line argument, which defaults to support for the VT100, determines the screen driver.

**sortname**

*sortname* sorts a list of person’s names by the last names.
splitlit creates a string literal with continuations in case it’s too long. Option -w i specifies the width of piece on line, default 50. Option -i i specifies the indent, default 3.

streamer

streamer outputs one long line obtained by concatenating the lines of the input file. Option -l i stops when line reaches or exceeds i; default no limit. Option -s s inserts separator s after each line; default no separator. Separators are counted in the length limit.

strpsgml [-f translation-file] [left-delimiter [right-delimiter]] (RLG)

strpsgml strips or translates SGML <>-style tags. The default left-delimiter is <, the default right delimiter is >. If no translation file is specified, the program strips material between the delimiters. The format of the translation file is:

```
code initialization completion
```
A tab or colon separates the fields. If you want to use a tab or colon as part of the text (and not as a separator), place a backslash before it. The completion field is optional.

tablc

tablc tabulates characters and lists each character and the number of times it occurs. Characters are written using escapes.

tablw

tablw tabulates words and lists number of times each word occurs. A word is defined to be a string of consecutive upper- and lowercase letters with at most one interior occurrence of a dash or apostrophe.

textcnt

textcnt tabulates the number of characters, "words", and lines in standard input and gives the maximum and minimum line length.

textcvt (RJA)

textcvt converts text file(s) among various platforms’ formats. The supported text file types are UNIX, MS-DOS, and Macintosh. The files are either converted in-place by converting to a temporary file and copying the result back to the original, or are copied to a separate new file, depending on the command line options.

toktab

toktab reads the token files given on the command line and summarizes them in a single file. Option -n sorts tokens by category in decreasing numerical order; default is alphabetical. Option -l i limits output in any category to i items; default no limit

trim [n]

trim copies lines from standard input to standard output, truncating the lines at n characters and removing trailing blanks. The default for n is 80.

ttt (CT)

ttt plays the game of tic-tac-toe.
turing

turing simulates the operation of an n-state Turing machine. The machine starts in state 1 with an empty tape. A description of the Turing machine is read from the file given as a command-line argument, or from standard input if none is specified. Comment lines beginning with ‘#’ are allowed, as are empty lines. The program states must be numbered from 1 and must appear in order. Each appears on a single line in this form:

\begin{verbatim}
sss. wdn

end
\end{verbatim}

\texttt{sss} is the state number in decimal. The \texttt{wdn} fields specify the action to be taken on reading a 0 or 1 respectively: \texttt{w} is the digit to write (0 or 1), \texttt{d} is the direction to move (L/l/R/r, or H/h to halt), \texttt{nn} is the next state number (0 if halting).

unique

\texttt{unique} filters out (deletes) identical adjacent lines in a file.

unpack

\texttt{unpack} unpackages files produced by \texttt{pack.icn}. See that program for limitations.

upper

\texttt{upper} maps the names of all files in the current directory to uppercase.

verse

This verse maker was initially published in an early 1980s Byte magazine in TRS80 Basic. It fetches the vocabulary from a vocabulary file specified on the command line; it looks for \texttt{verse.dat} by default. See that file for examples of form.

versum

\texttt{versum} writes the versum sequence for an integer to a file of a specified name. If such a file exists, it picks up where it left off, appending new values to the file.

webimage

\texttt{webimage} takes image filename arguments and writes a Web page that embeds each image. Option \texttt{-a s} specifies alignment, default "bottom". Option \texttt{-t s} supplies a title for the page; default "untitled". Option \texttt{-n} directs to include file names.

what

\texttt{what} writes all strings beginning with "@" followed by "(\#)" and ending with null, newline, quotes, greater-than or backslash. Follows UNIX \texttt{what(1)} conventions.

when

\texttt{when} is like a date based \texttt{ls} command. UNIX \texttt{find} can do the same things, but \texttt{find} is a bit arcane, so \texttt{when} provides a simpler alternative. Many options and clauses are supported. Look at the code for clues. This one only works in the current directory. Requires: UNIX.

xtable

\texttt{xtable} prints tables for ASCII, EBCDIC and their decimal and octal values.
APPENDIX B. THE ICON PROGRAM LIBRARY

yahtz

A classic "dice poker" game.

zipsort

zipsort sorts labels produced by labels in ascending order of their postal zip codes. Option -d n sets the number of lines per label to n (default 9). This value must agree with that used to format the labels. The zip code must be the last nonblank string at the end of the label. It must consist of digits but may have an embedded dash. See also: labels.icn.

B.3 Selected IPL Authors and Contributors

This Appendix presents the work of the following authors and contributors of Icon Program Library modules and programs. Ralph Griswold deserves primary credit for initiating and maintaining the collection. To a great extent, the various authors’ code is described in their own words, from the public domain documentation they have written about it. We would like to acknowledge their contribution to the Icon community and to this book. Any errors that remain in this Appendix are solely our responsibility, and we apologize for them.

Paul Abrahams    Robert J. Alexander    Allan J. Anderson
Norman Azadian  Alan Beale            Phil Bewig
Peter A. Bigot  David S. Cargo        Alex Cecil
Alan D. Corre   Cary A. Coutant       William E. Drissel
Erik Eid        Ronald Florence       David A. Gamey
Michael Glass   Richard L. Goerwitz    Ralph E. Griswold
Matthias Heesch  Charles Hethcoat     Anthony V. Hewitt
Thomas R. Hicks Clinton L. Jeffery    Jere K?p?yaho
Justin Kolb     Tim Korb              Frank J. Lhota
Nevin J. Liber  William P. Malloy     C. Scott McArthur
Will Menagarini Joe van Meter         William H. Mitchell
Rich Morin      Jerry Nowlin          Mark Otto
Robert Parlett  Jan P. de Ruiter      Randal L. Schwartz
Charles Shartsis David J. Slate       John D. Stone
Chris Tenaglia  Phillip L. Thomas     Gregg M. Townsend
Kenneth Walker  Stephen B. Wampler    Beth Weiss
Robert C. Wieland Cheyenne Wills      David Yost
Appendix C

The Unicon Component Library

C.1 GUI Classes

This section presents the various methods and class fields making up the classes implemented in the Unicon GUI library. The library is a package (gui) defined by a set of files in their own directory (uni/gui). In this section if a class has superclasses, their names are given, separated by colons. Superclasses should be consulted for additional methods and class fields used by subclasses. The default is to have no superclass. If a required field is omitted, an error message may be produced, such as:

    gui.icn : error processing object TextList : x position unspecified

This means that the x position of a TextList object was not specified, probably because the \texttt{set\_pos()} method had not been invoked.

You will generally need to consult "Graphics Programming in Icon", by Griswold, Jeffery, and Townsend in order to make the best use of these classes.

Event

An instance of this class is generated by components and passed to the connected event handler method. It holds three elements, which are accessed as follows:

\texttt{get\_source()} returns the component associated with the event. This may be a subclass of either \texttt{Component} or a \texttt{MenuComponent}. If this is \texttt{&null}, then an Icon event has occurred which has not produced an Event from any component.

\texttt{get\_type()} returns a field to distinguish between different types of event generated by the same component (\texttt{ACTION\_EVENT}, \texttt{MOUSE\_PRESS\_EVENT}, etc.).

\texttt{get\_param()} returns the parameter associated with the event, if any.

Dialog : Container

This is the parent class of a dialog window.

\texttt{resize\_win(w, h)} resizes the window to the given dimensions.

\texttt{get\_win()} returns the Icon window associated with the dialog.
set_min_size(w, h) sets the minimum dimensions for a window. The user will not be able to resize the window below this size.

set_focus(c) sets the keyboard focus to component c. clear_focus() clears the focus.

dialog_event(e:Event) must be defined in the subclass. It is invoked on each event.

dispose(x) is normally invoked in response to an event. It sets a flag to indicate that the dialog should be closed. If x is non-null, the window is closed already.

set_unique(c) is called by a component c to indicate the beginning of unique event processing, whereby one component alone receives all events.

    clear_unique(x) unsets unique event processing mode. If x is &null then the final event in unique processing mode will be passed to all objects; otherwise it will not.

    show_modal(d) opens and displays the dialog window, and accepts events into it. As events occur they are handled, until a call to dispose() is made, then the window is closed and the method returns. Parameter d is the parent dialog, if any.

    Open() opens the dialog window. Close() closes the dialog window.

    process_event(e) processes Icon event e, calling event handlers.

    win : window is the dialog's window.

is_open : flag indicates whether the window is open.

focus : Component specifies the component with the current focus.

unique_flag : flag controls unique processing, where one component receives all events.

re_process_flag : flag tells whether to distribute last Icon event during unique mode.

buffer_win is a buffer window for double buffering.

min_width : integer is the minimum width of window, or &null if no minimum.

min_height : integer is the minimum height of window, or &null if no minimum.

**Component**

This is the parent class of all the GUI components. All of its methods and variables therefore apply to its sub-classes.

    get_x_reference(), get_y_reference(), get_w_reference(), and get_h_reference() produce this object's x position, y position, width, and height values from which to compute absolute positions. May be over-ridden; by default returns self.x, self.y, self.w, self.h.

    get_cwin_reference(), get_cbwin_reference() produce the cloned window (the reference object inherits the attributes by cloning this window) and cloned buffer window.

    fatal(s) prints an error message s together with the class name, and stops the program.

    generate_components() : Component* generates the components for the object. By default this just returns the component itself.

    is_shaded() : flag? succeeds if the component is shaded. A shaded component, such as a button, may be displayed differently, and will not generate events.

    set_is_shaded() sets the shaded status of the component to shaded.

    clear_is_shaded() sets the shaded status of the component to not shaded.

    toggle_is_shaded() swaps the shaded status of the component.

    set_draw_border() sets the component such that a border is drawn.
clear_draw_border() sets the component such that a border is not drawn.

toggle_draw_border() toggles whether or not to draw a border around the component. Different objects respond differently to this flag being set; some ignore it altogether.

display(buffer_flag) draws, or re-draws, the component in the dialog window. If buffer_flag is not null, then the component is displayed into the buffer window, not the dialog window (this is used for double-buffering purposes).

set_accepts_tab_focus() sets the accepts_tab_focus_flag, meaning that the object will gain the keyboard focus by way of the user pressing the tab key repeatedly.

clear_accepts_tab_focus() clears the accepts_tab_focus_flag.

attrib(x[]) sets the graphic attributes of the component to the given parameter list. For example: w.attrib("font=helvetica", "bg=pale blue"). Arguments may be either strings or lists of strings. A description of the attributes is in Chapter 7 and detailed in [GJT98].

get_parent_win() : window returns the window in which the component resides.

set_pos(x, y) sets the x and y position of the component. Each coordinate can be either an absolute pixel position, or a percentage plus or minus an offset.

c.set_pos(100, "25\%")

c.set_pos("50\%-20", "25\%+100")

set_align(x_align:"l", y_align:"t") sets the alignment of the component. Options for x_align are "l", "c" and "r", for left, center, and right alignment. Options for y_align are "t", "c" and "b", for top center and bottom alignment. Examples:

# Place c centered in the center of the window,
# its top left corner at (10,10).
c.set_pos("50\%", "50\%")
c.set_align("c", "c")
c.set_pos(10, 10)
c.set_align("l", "t")

set_size(w, h) sets the size of the component; parameters are as for set_pos() above. Some components have sensible default sizes, but on others the size must be set explicitly.

handle_event(e) : Event? is over-ridden by all this class’s subclasses. It is the method that handles an Icon event e. It would not normally be called by a user program. Its result is passed to the dialog_event() method of the dialog.

The first two fields of the Event structure are the Icon event e and the object itself. The third field is the code, which can be any integer. For example:

    method handle_event(e)
        ...
        return Event(e, self, 0)
    end
do_shading(w) is called from a component’s display() method. This method filters the component to give a shaded appearance if the is_shaded_flag is set. w is the window to use (normally cwin).

in_region(x:&x,y:&y) succeeds for unshaded components if (x,y) lies within the component.

got_focus() is called when the component gets the keyboard focus. It may be extended in subclasses. For example:

```
method got_focus()
    self$Component.got_focus()
    #
    # Display the box cursor
    #
    display()
end
```

lost_focus() is called when the component loses the keyboard focus; it may be extended.

unique_start() initiates unique event processing for this object by calling the parent dialog’s set_unique() method.

unique_end(x) ends unique event processing for this object by calling the parent dialog’s clear_unique(x) method.

init() sets up the cloned windows for the component. This method should be called for any components created and used inside any custom components.

set_parent_dialog(x) sets the owning Dialog of the component to x.

get_parent_dialog() : Dialog returns the parent dialog of the component.

firstly() is invoked after the position of the object has been computed, but before the object has been displayed in the window. This method may be extended in subclasses.

finally() is invoked before the window is closed; it may be extended in subclasses:

```
method finally()
    self$Component.finally()
    # Do something here
    ...
    return
end
```

resize() computes the absolute positions and sizes from the specifications given by set_pos() and set_size(). This method needs to be extended for a component that contains other components. See the section on custom components for an example.

x_spec, y_spec are the x and y positions as specified by set_pos(), e.g. "50%"

w_spec, h_spec are the width and height specifiers as used in set_size(), e.g. "100%"

x_align, y_align are the x and y alignment as specified in set_align(), e.g. "l", "b".
The following four attributes are absolute dimensions in pixels, compiled from \texttt{x\_spec}, \texttt{y\_spec}, \texttt{w\_spec}, and \texttt{h\_spec}, and the dimensions of the enclosing object or window.

\texttt{x} and \texttt{y} are the \texttt{x} and \texttt{y} positions computed from \texttt{x\_spec} and \texttt{y\_spec}.

\texttt{w} and \texttt{h} are the width and height computed from \texttt{w\_spec} and \texttt{h\_spec}.

\texttt{cwin} is a cloned window created by combining the Dialog's canvas with the Component's attributes, so drawing into this window will draw straight to the Dialog window with the correct attributes. \texttt{cbwin} is a cloned window created by combining a buffer window with the Component's attributes. This is used solely for double-buffering purposes.

\texttt{parent\_dialog} is the Dialog class instance of which this Component is a part.
\texttt{attrs} is a list of strings specifying graphics attributes, e.g. \texttt{["bg=blue", "resize=on"]}.
\texttt{has\_focus: flag} indicates whether the Component currently has the keyboard focus.
\texttt{accepts\_tab\_focus\_flag: flag} indicates whether the Component accepts keyboard focus by way of the tab key being pressed.
\texttt{draw\_border\_flag: flag} indicates whether the Component should have a border drawn around it. Many components (such as TextButtons) ignore this flag.
\texttt{is\_shaded\_flag: flag} indicates whether the Component currently is shaded.
\texttt{reference} links to the object used to calculate absolute sizes from percentage sizes. For objects placed directly in the Dialog, rather than in some other object, this will point to the Dialog itself, which over-rides the several methods \texttt{get\_x\_reference()} etc., appropriately.

\textbf{Container : Component}

This class contains other components. The container itself is invisible. Many of \texttt{Component}'s methods are over-ridden by \texttt{Container}. A \texttt{Dialog} is a sub-class of this class.
\texttt{add(c:Component)} adds the component \texttt{c} to the Container.
Instance variable \texttt{components} are the components inside the Container.

\textbf{VisibleContainer : Component}

This is similar to a Container, except that the object itself is a capable of display.
\texttt{add(c:Component)} adds the component \texttt{c} to the VisibleContainer.
components are the components inside the VisibleContainer.

\textbf{Button : Component}

This is the parent class of button classes including TextButton and IconButton. A button produces an Event of code 0 when the button is depressed, and code 1 when it is released. By default, when a button holds the keyboard focus a dashed line appears just within the button. When return is pressed an event of code 2 is generated. The method \texttt{Dialog.set\_initial\_focus()} can be used to give the button the keyboard focus when the dialog is first displayed.

\texttt{set\_no\_keyboard()} disables the keyboard control over the button described above. No dashed line will ever appear in the button display and return will have no effect on the button even if it has the focus.

\textbf{TextButton : Button}

A button with a text label. The size of the button can either be set using \texttt{set\_size()} or
be left to default to a size based on the given label.

    set_internal_alignment(x) sets the alignment of the label within the button. The parameter should be either "l", "c" or "r" to set the alignment to left, center or right respectively. If this method is not invoked, then the alignment is centered.

    set_label(x) sets the label in the button to the given string. Examples:

        b := TextButton()
b.set_label("Cancel")
b.set_pos("50\%", "80\%")
b.set_align("c", "c")
add(b)

**IconButton : Button**

This is a button with an Icon image within it. There is a useful program in the Icon program library called xpmtoims, which will take an xpm file and output the equivalent Icon image string, which can then be inserted into a program. See also the X Window programs sxpm and pixmap for viewing and editing xpm files respectively.

A border may be requested with set_draw_border(). Unless explicitly specified, the size will default to the image's size, plus a standard surrounding area if a border is requested.

    set_img(s:string) sets the image to s, which should be in Icon image format. Examples:

        # Create a button with a diamond image and a border
        b := IconButton()
b.set_draw_border()
b.set_img("11,c1,_
           0-----
           000---
           0000---
           00000---
           000000---
           0000000---
           00000000---
           000000000---
           0000000---
           00000---
           0-----
        ")

**ButtonGroup**

This class groups several Buttons together. Then, when the mouse is clicked down on one of the Buttons and then dragged onto another before being released, the other Button will go "down". This is the common behavior for buttons in a bar along the top of an application.

Note: a Button must be added to both the ButtonGroup and the Dialog too. Examples:

        bg := ButtonGroup()
b := TextButton()
add(c:Button) adds the given Button to the ButtonGroup.

Label : Component
This simply creates a text label in the dialog window. Calling set_draw_border() adds a border around the label. The size will default if not set.

set_label(s:string) sets the label to the given string.

set_internal_alignment(x) sets the horizontal alignment of the label within the area of the component; should be "l", "c" or "r". Default is "l". If the horizontal size is left to default, then setting this field should make no difference, because the size of the component will be set so that the string just fits into it.

Icon : Component
This displays an icon, supplied in Icon image format. A border may be requested with set_draw_border(). The size defaults to the image’s size, plus a standard surrounding area if a border is requested. set_img(s:string) sets the image to be displayed.

Image : Component
This class loads an image from a file and displays it. The image should be in GIF format. A border may be included with set_draw_border(). The size of the area into which the image is drawn must be set with set_size().

set_filename(s:string) sets the name of the file from which to load the image; redisplay the image from the new file if appropriate.

set_scale_up() scales the image up to fit in the space specified by set_size(). The image is not distorted, but will be expanded to fill one of the dimensions depending on its shape. If the image is bigger than the specified size then it will be scaled down.

set_internal_alignment(x:"c", y:"c") sets the horizontal and vertical alignment of the image within the component; x should be "l", "c" or "r", y should be "t", "c" or "b".

Border : VisibleContainer
This class provides decorative borders. The add(c) method may optionally be used to set one other component to be the title of the Border. This would normally be a Label object, but it could also be a CheckBox, Icon, or whatever is desired.

set_internal_alignment(x) sets the alignment of the title to "l", "c" or "r".

b := Border()
#
# Add a Label as the title
#
I := Label()
I.set_label("Title String")
b.add(I)
add(b)
ScrollBar : Component

This class provides horizontal and vertical scroll bars. The first way to use a scroll bar is to set a total_size (represented by the whole bar), a page_size (represented by the draggable button) and an increment_size (being the amount added/subtracted when the top/bottom button is pressed). The value then ranges from zero to (total_size - page_size) inclusive. An initial value must be set with the set_value() method. For example:

```lisp
vb := ScrollBar()
vb.set_pos("85\%", "25\%")
vb.set_size(20, "40\%")
vb.set_total_size(130)
vb.set_page_size(30)
vb.set_increment_size(1)
vb.set_value(0)
add(vb)
```

Alternatively, a scroll bar can be used as a slider over a given range of values. In this case, the range is set with set_range(). It is still necessary to set the increment_size and the initial_value, but page_size and total_size should not be set. Real numbers as opposed to integers can be used for the range settings if desired. For example:

```lisp
vb := ScrollBar()
vb.set_pos("85\%", "25\%")
vb.set_size(20, "40\%")
vb.set_range(2, 25)
vb.set_value(10)
vb.set_increment_size(1)
add(vb)
```

An Event is returned whenever the buttons are pressed or the bar dragged; the value can be retrieved by get_value(). The event code (obtainable by get_code()) is 1 if the bar has been dragged, and 0 if either button has been pressed or the bar released after being dragged. This fact can be used to reduce the number of events which are processed by the user’s program - just ignore events with code 1.

- set_is_horizontal() makes the scroll bar horizontal (default is vertical).
- set_range(x, y) sets the scroll bar range (integer or real) from x to y inclusive.
- set_total_size(x) sets the total size which the scroll bar area represents.
- get_total_size() returns the total size.
- set_page_size(x) sets the size that the bar in the scroll bar area represents.
- get_page_size() gets the page size.
- set_value(x) sets the value representing the top of the bar in the scroll bar. The value is forced into range if it is not in range already.
- get_value() gets the value.
set_increment_size(x) sets the amount by which to increase when a button is pressed.

**TextField : Component**

TextField is a class for a single input line of text. The text can scroll within the area specified. By default, a border surrounds the text area; this can be turned off by using clear_draw_border(). The horizontal size must be set by the set_size() method: there is no default (the vertical size will default, however). An event is generated when return is pressed (with code 0), and whenever the contents are changed (with code 1).

get_contents() returns the present contents of the text field. set_contents(x) sets the contents of the field. If not invoked then the initial content is the empty string. Examples:

```plaintext
t := TextField()
t.set_pos(50, 250)
# Vertical size will default
t.set_size(100)
t.set_contents("Initial string")
add(t)
```

**CheckBox : Component**

This class creates a small button with a label which is either in an on or off state. The button is an Icon image, which may be specified by the user if desired. The images and size default to appropriate values if not specified.

set_imgs(x, y) sets the up/down images for the button. The images should be in Icon image format. The two images must have the same dimensions.

is_checked() succeeds if the button is down (checked); fail otherwise.

toggle_is_checked() toggles the initial status of the button.

set_is_checked() sets the status of the button to checked.

clear_is_checked() sets the status of the button to not checked.

set_label(x) sets the label of the component to the given string.

get_status() returns 1 if the CheckBox is checked, &null otherwise. Examples:

```plaintext
c := CheckBox()
c.set_pos(200, 100)
c.set_label("Checkbox")
add(c)
```

**CheckBoxGroup**

This class contains CheckBox objects that act together as "radio buttons". The image style of CheckBoxes in a CheckBoxGroup draws diamonds rather than boxes. The status of a CheckBoxGroup should be set with the set_which_one() method, not by turning the individual CheckBoxes on/off with their own methods - that would confuse the program. Note: a CheckBox must be added to both the CheckBoxGroup and the dialog box.

set_by_flag(i) sets the CheckBox which is down according to the integer i. If i = 1 then the first CheckBox is down, if i = 2 the second is down, etc for i = 4, 8, 16.
**get_by_flag()** returns an integer in the range 1, 2, 4, 8 ... depending upon whether the first, second, third etc CheckBox is down.

**add(c:CheckBox)** adds c to the CheckBoxGroup.

**get_which_one()** returns the CheckBox which is currently down.

**set_which_one(x:CheckBox)** sets which CheckBox is down to x. Examples:

```plaintext
#  # Create a CheckBoxGroup of 3 CheckBoxes  #
c := CheckBoxGroup()
c1 := CheckBox()
c1.set_pos(200, 50)
c1.set_label("Checkbox 1")
add(c1)
c.add(c1)

c2 := CheckBox()
c2.set_pos(200, 90)
c2.toggle_is_shaded()
c2.set_label("Checkbox 2")
add(c2)
c.add(c2)

c3 := CheckBox()
c3.set_pos(200, 130)
c3.set_label("Checkbox 3")
add(c3)
c.add(c3)

#  # Initially, set the first one "on"  #
c.set_which_one(c1)
```

**TextList : Component**

This class displays a list of strings. The list is optionally editable, providing a multiline text input region. If in editable mode then an event is generated whenever the contents change; otherwise an event is generated when a line is highlighted by being clicked. Horizontal and vertical scroll bars are displayed if necessary.

If not in editable mode, then optionally the user can be allowed to select either one line only, or several lines. In either case, an event is generated when a line is selected. There is quite a high computational overhead for an editable TextList. For a large number of lines, this may result in a sluggish performance.

**set_is_editable()** specifies that the contents may be edited.
get_contents() : list returns the current contents as a list of strings.

set_contents(x:list, line, left_pos, preserve_selections) sets the contents to x and sets the position to line and left_pos. If these parameters are omitted then the default is to start at line 1, with left offset zero if the window is not already open, or to retain the existing position if it is. If the last parameter is non-null then the current selections are retained; otherwise they are reset. This method has no effect if the component is in editable mode and the window is already open.

```
tl := TextList()
tl.set_contents(data)

# Amend data and go to end of data
put(data, "New line")
tl.set_contents(data, *data, 0)
```

The method set_select_one() specifies that only one line of the list may be highlighted, while method set_select_many() specifies that several lines of the list may be highlighted. Of no effect if in editable mode.

get_selections() : list returns a list of the numbers of the lines that are highlighted.

set_selections(L) sets the highlighted selections to a given list of line numbers.

```
tl := TextList()
tl.set_pos("50%", "50%")
tl.set_size("70%", "50%")
tl.set_align("c", "c")
tl.set_contents(data) # data is a list of strings
add(tl)
```

DropDown

This class is a superclass of List and EditList below.

set_selection_list(x) sets the list of selections to the list x.

get_selection():integer returns the index of the item in the list presently selected.

List : Component : DropDown

This component is for selecting one string from a list. When a button is pressed a list appears (possibly with a scroll bar) from which one item can be selected. An Event is generated whenever an item is selected. A width must be specified for this component.

set_selection(x) sets the selected item to element x.

set_constant_label(x) supplies a string that will always appear in the text part of the component, rather than the currently selected item.

The methods for handling the list of selections are mostly inherited from DropDown.

```
L := List()
L.set_selection_list(["Red","Green","Yellow","Blue","Orange"])
```
L.set_size(120)
L.set_pos(100, 100)
L.set_selection(2)  # Green will be the first selection
add(L)

**EditList : Component : DropDown**

An **EditList** works like a **List**, but the user may edit the item that is selected. An extra method is therefore supplied to get the content, as it may not correspond to an element of the list. An **Event** is generated with code 0 if an element of the list is selected, with code 1 if return is pressed, and with code 2 if the user edits the selected item.

- **get_contents()** returns the contents of the selected item (which may have been edited).
- **set_contents(x)** sets the initial contents of the text to the given string.

**MenuBar : Component**

This class is the base from which menu systems are created. **Menu items** are added to this class; they are not separate components added to the dialog itself. The default position is (0, 0); the default size is 100% of the width of the screen and a reasonable height based on the font specified. **add(c:Menu)** adds c to the MenuBar. This will be one drop down menu. Items are then added to the **Menu**.

**MenuButton : Component**

This is similar to MenuBar, but holds just a single drop-down menu, rather than several.

- **set_menu(x:Menu)** sets the menu to be displayed when the component is clicked.

**MenuEvent : Event**

This subclass adds variable **menu_code**, returned by a **MenuComponent**, indicating to the menu system what to do with the event. It takes the following values:

<table>
<thead>
<tr>
<th>Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAIL_1</td>
<td>The Menu system closes and the last event is not passed on to the dialog.</td>
</tr>
<tr>
<td>FAIL_2</td>
<td>The Menu system closes and the last event is passed on to the dialog.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>The Menu system does not close and continues to process events.</td>
</tr>
<tr>
<td>SUCCEED</td>
<td>The menu system closes and produces the given event, which is passed to the dialog’s <code>dialog_event()</code> method. The last event is not passed to other components.</td>
</tr>
</tbody>
</table>

**MenuComponent**

This is the superclass of all the objects that make up the menu system (other than MenuBar of course). For components that appear in a menu with a label, an optional left/right string/image can be set.

- **set_label_left(x)** sets the optional left label to the given string.
- **set_label_right(x)** sets the optional right label to the given string.
- **set_img_left(x)** sets the optional left image to the given Icon image.
- **set_img_right(x)** sets the optional right image to the given Icon image.
- **toggle_is_shaded()** toggles whether or not the item is shaded. If it is, it is displayed in a filtered way and will not accept input.
- **set_is_shaded()** sets the shaded status of the component to shaded.
clear_is_shaded() sets the shaded status of the component to not shaded.
set_label(x) sets the center label to the given string.

SubMenu : MenuComponent
This class encapsulates a Menu object that when selected will display something outside
the menu itself (for example a sub-menu of other menu items). It is intended to be extended
by custom menu components, and should not be instantiated. Methods are empty.
hide_non_menu() and set_which_open(x) are called by Menu for any SubMenu object,
but would not normally need to be over-ridden by a custom class.
resize() may be over-ridden to initialize the width and height of the object.
display() must be over-ridden; it displays the object.
handle_event(e) handles the Icon event e and must be over-ridden.
hide() hides (closes) the object’s display and must be over-ridden.

Menu : SubMenu
This class encapsulates a drop down menu, or a sub-menu. The left, center and right
labels/images of the elements within it are formatted within the menu automatically.
add(c:Component) adds the given component to the Menu.

TextMenuItem : MenuComponent
This class encapsulates a single text item in a Menu. It has no additional methods that
the user need call other than are contained in its parent class, MenuComponent.

CheckBoxMenuItem : MenuComponent
This class encapsulates a check box in a menu. Several CheckBoxMenuItems may be
added to a CheckBoxGroup structure to give "radio buttons" within menus.
set_imgs(x, y) sets the up and down images to x and y respectively. The default is
boxes, unless the component is in a CheckBoxGroup in which case the default is diamonds.
is_checked() succeeds if the component is checked; fail otherwise.
set_is_checked() sets the status of the button to checked.
clear_is_checked() sets the status of the button to not checked.

MenuSeparator : MenuComponent
This is a horizontal bar in a Menu, for decorative purposes. It has no user methods.

TableColumn : TextButton
This class provides one column within a Table class, which displays a table of data. A
column has a label with a button that produces an event when clicked. The column may be
expanded or contracted by dragging the right edge of the button. Calling the set_label(x)
method of the superclass, TextButton, sets the label.
set_column_width(x) sets the initial width of the column in pixels; this is required.

c1 := TableColumn()
c1.set_internal_alignment("r") # Label is right aligned
c1.set_column_width(80)
c1.set_label("Number")
Table : Component
This class displays a table, the columns of which are set up using TableColumns.

set_button_bar_height(x) sets the height of the buttons at the top in pixels.
set_contents(x) sets the contents of the table. The parameter should be a two
dimensional list. Each element of the list should correspond to one row of the table.
set_contents(x, line:1, left_pos, preserve_selections) sets the contents to x and sets
the position to line and left_pos. The default left_pos offset is the existing position if
the window is already open, or zero otherwise. The last parameter is a flag that directs
whether to retain the current selections.

add(c:TableColumn) adds the given TableColumn to the Table.
set_select_one() specifies that only one row of the table may be highlighted.
set_select_many() allows several rows of the table to be highlighted.
get_selections() returns a list of the numbers of the rows which are highlighted.
set_selections(L) sets the line numbers that are selected to the list of line numbers L.

TabItem : Container
This class represents a single pane in a TabSet. Components are added to the TabItem
using Container’s add() method. They are then displayed and accept input when that
TabItem is selected. Components added to the TabItem are positioned relative to the
position and size of the parent TabSet. Therefore for example set_pos("50\%", "50\%")
refers to the center of the TabSet rather than the center of the screen. The components
also inherit any window attributes of the TabSet, such as font, color and so on.

set_label(x) sets the TabItem’s label.

TabSet : VisibleContainer
This class holds the several TabItems.
set_which_one(x) sets the displayed TabItem; default is the first one.
add(c:TabItem) adds the given TabItem to the TabSet.

Panel : VisibleContainer
This class simply contains other components. The components inside have their sizes
and positions computed relative to the Panel and also inherit the Panel’s windowing at-
tributes. Components are added using the add() method of VisibleContainer.

OverlayItem : Container
This class is one "pane" in an OverlaySet, which is rather like a TabSet except that
there are no tabs, and control over which pane is displayed is entirely the affair of the
program. The components inside have their sizes and positions computed relative to the
parent OverlaySet and also inherit the OverlaySet’s windowing attributes. Components
are added using the add() method of Container.

OverlaySet : VisibleContainer
An OverlaySet is a set of OverlayItems.
set_which_one(x) sets the currently displayaed OverlayItem; default is the first.
add(c:OverlayItem) adds the given OverlayItem to the OverlaySet.
Appendix D

Differences between Icon and Unicon

This appendix summarizes the known differences between Arizona Icon and Unicon. Since the language has myriad additions covered by the whole book, the emphasis of this page is on incompatibilities that might require changes to existing Icon programs.

D.1 Extensions to Functions and Operators

Unicon broadens the meaning of certain pre-existing functions where it is consistent and unambiguous to do so. These extensions revolve primarily around the list type. For example, `insert()` allows insertion into the middle of a list, `reverse()` reverses a list, and so forth.

D.2 Objects

Unicon supports the concepts of classes and packages with declaration syntax. This affects scope and visibility of variable names at compile time. At runtime, objects behave similar to records in most respects. These additions include reserved words that are no longer valid variable names, such as `class`, `package`, and `import`.

D.3 System Interface

Unicon’s system interface presumes the availability of hierarchical directory structure, communication between programs using standard Internet protocols, and other widely available facilities not present in Arizona Icon.

Unicon’s graphics include extensive 3D facilities. The 2D facilities are extended with additional image file formats.
D.4 Database Facilities

Unicon supports GDBM and SQL databases with built-in functions and operators. The programmer manipulates data in terms of persistent table and record abstractions. SQL database support may not be present on platforms that do not provide ODBC open database connectivity drivers.

D.5 Multiple Programs and Execution Monitoring Support

Unicon virtual machine interpreters by default support the loading of multiple programs so that various debugging and profiling tools can be applied to them without recompilation. The execution monitoring facilities are described in "Program Monitoring and Visualization: An Exploratory Approach", by Clinton Jeffery. Unicon optimizing compilers may omit or substitute for these facilities.
Appendix E

Portability Considerations

Unicon’s POSIX-based system interface facilities presented in Chapter 5 are portable. You can expect the portable system interface to be available on any implementation of Unicon. This appendix presents additional, non-portable elements of the Unicon POSIX interface, as well as some notes on functionality specific to Microsoft Windows.

E.1 POSIX extensions

The extensions presented in this section may or may not be part of the POSIX standard, but they are a part of the Unicon language as implemented on major POSIX-compliant UNIX platforms such as Solaris and Linux. Ports of Unicon to non-POSIX or quasi-POSIX platforms may or may not implement any of these facilities.

Information from system files

There are four functions that read information from the system: getpw() to read the password file, getgr() for the group file, gethost() for hostnames, and getserv() for network services. Called with an argument (usually a string), they perform a lookup in the system file; called with no arguments, these functions step through the files one entry at a time.

The functions setpwent(), setgrent(), sethostent(), and setservent() do the same things as their POSIX C language counterparts; they reset the file position used by the get* routines to the beginning of the file. These functions return records whose members are similar to the C structures returned by the system functions getpwuid(2), gethostbyname(2), etc.

fork and exec

POSIX-compliant systems support a process-launch interface using the functions fork() and exec(). fork() makes a copy of the current process. After the fork there are two identical processes that share resources such as open files, and differ only in one respect: the return
value they received from the call to `fork()`. One process gets a zero and is called the child; the other gets the process id of the child and is called the parent.

Usually `fork()` is used to run another program. In that case, the child process uses the system call `exec()` which replaces the code of the process with the code of a new program. This `fork()/exec()` pair is comparable to calling `system()` and using the option to not wait for the command to complete.

The first argument to `exec()` is the filename of the program to execute, and the remaining arguments are the values of `argv` that the program will get, starting with `argv[0]`.

```plaintext
exec("/bin/echo", "echo", "Hello,", "world!")
```

**POSIX functions**

These functions are present in all Unicon binaries, but you can expect them to fail on most non-UNIX platforms. Check the readme.txt file that comes with your installation to ascertain whether it supports any of these functions.

- `chown(chown(f, u:-1, g:-1) : ? change owner` chown(f, u, g) sets the owner of a file (or string filename) f to owner u and group g. The user and group arguments can be numeric ID’s or names.

- `chroot(string) : ? change filesystem root` chroot(f) changes the root directory of the filesystem to f.

- `crypt(string, string) : string encrypt password` crypt(s1, s2) encrypts the password s1 with the salt s2. The first two characters of the returned string will be the salt.

- `exec(string, string, ...) : null execute program` exec(s, arg0, arg1, arg2, ...) replaces the currently executing Icon program with a new program named in s. The other arguments are passed to the program. Consult the POSIX exec(2) manual pages for more details. s must be a path to a binary executable program, not to a shell script (or, on UNIX) an Icon program. If you want to run such a script, the first argument to exec() should be the binary that can execute them, such as /bin/sh.

- `fcntl(file, string, options) file control` fcntl(file, cmd, arg) performs miscellaneous operations on the open file. See the fcntl(2) manual page for more details. Directories and DBM files cannot be arguments to fcntl(). The following characters are the possible values for cmd:

  - f Get flags (F_SETFL)
  - F Set flags (F_GETFL)
  - x Get close-on-exec flags (F_GETFD)
  - X Set close-on-exec flag (F_SETFD)
E.1. POSIX EXTENSIONS

1. Get file lock (F_GETLK)
L. Set file lock (F_SETLK)
W. Set file lock and wait (F_SETLKW)
o. Get file owner or process group (F_GETOWN)
O. Set file owner or process group (F_SETOWN)

In the case of L, the arg value should be a string that describes the lock, otherwise arg is an integer. A record `posix_lock(value, pid)` will be returned by F_GETLK.

The lock string consists of three parts separated by commas: the type of lock (r, w or u), the starting position, and the length. The starting position can be an offset from the beginning of the file (e.g. 23), end of the file (e.g. -50), or from the current position in the file (e.g. +200). A length of 0 means lock till EOF. These characters represent the file flags set by F_SETFL and accessed by F_GETFL:

d. FNDELAY
s. FASYNC
a. FAPPEND

fdup(file, file) : ?
duplicate file descriptor

`fdup(src, dest)` is based on the POSIX `dup2(2)` system call. It is used to modify a specific UNIX file descriptor, such as just before calling `exec()`. The dest file is closed; src is made to have its Unix file descriptor; and the second file is replaced by the first.

filepair() : list
create connected files

`filepair()` creates a bi-directional pair of files analogous to the POSIX `socketpair(2)` function. It returns a list of two indistinguishable files; writes on one will be available on the other. The connection is bi-directional, unlike that of function `pipe()`. Caution: typically, the pair is created just before a `fork()`; after it, one process should close L[1] and the other should close L[2] or you will not get proper end-of-file notification.

fork() : integer
fork process

`fork()` creates a new process that is identical to the current process except in the return value. The parent gets a return value that is the PID of the child, and the child gets 0.

getegid() : string
get effective group identity

`getegid()` produces the effective group identity (gid) of the current process. The name is returned if it is available, otherwise the numeric code is returned.

geteuid() : string
get effective user identity

`geteuid()` produces the effective user identity (uid) of the current process. The name is returned if it is available, otherwise the numeric code is returned.

getgid() : string
get group identity

`getgid()` produces the real group identity (gid) of the current process. The name is returned if it is available, otherwise the numeric code is returned.
APPENDIX E. PORTABILITY CONSIDERATIONS

getgr(g) : record
get group information
getgr(g) returns a record that contains group file information for group g, a string group name or an integer group code. If g is null, each successive call to getgr() returns the next entry. setgrent() resets the sequence to the beginning. Return type:

    record posix_group(name, passwd, gid, members)

gethost(x) : record
get host information
gethost(n) for network connection n returns a string containing the IP number and port this machine is using for a network connection. gethost(s) returns a record that contains host information for the machine named s. If s is null, each successive call to gethost() returns the next entry. sethostent() resets the sequence to the beginning. The aliases and addresses are comma separated lists of aliases and addresses (in a.b.c.d format) respectively. Its return type is:

    record posix_hostent(name, aliases, addresses)

getpgrp() : integer
get process group
getpgrp() returns the process group of the current process.

g getpid() : integer
get process identification
g getpid() produces the process identification (pid) of the current process.

getppid() : integer?
g get parent process identification
g getppid() produces the pid of the parent process.

getpw(u) : posix_password
get password information
getpw(u) returns a record that contains password file information. u can be a numeric uid or a user name. If u is null, each successive call to getpw() returns the next entry and setpwent() resets the sequence to the beginning. Return type:

    record posix_password(name, passwd, uid, gid, age, comment, gecos, dir, shell)

getserv(string, string) : posix_servent
get service information
getserv(s, proto) returns a record that contains service information for the service s using protocol proto. If s is null, each successive call to getserv() returns the next entry. setservent() resets the sequence to the beginning. If proto is defaulted, it will return the first matching entry. Its return type is:

    record posix_servent(name, aliases, port, proto)

getuid() : string
get user identity
g getuid() produces the real user identity (uid) of the current process.

kill(integer, x) : ?
kil process
kill(pid, signal) sends a signal to the process specified by pid. The second parameter can be the string name or the integer code of the signal to be sent.

link(string, string) : ?
link files
link(src, dest) creates a (hard) link dest that points to src.

readlink(string) : string?
readlink(s) produces the filename referred to in a symbolic link at path s.

setgid(integer) : ?
setgid(g) sets the group id of the current process to g. See the UNIX setgid(2) man page.

setgrent() : null
setgrent() resets and rewinds the pointer to the group file used by getgr() when getgr() is called with no arguments.

sethostent(integer:1) : null
sethostent(stayopen) resets and rewinds the pointer to the host file used by gethost().
The argument defines whether the file should be kept open between calls to gethost(); a nonzero value (the default) keeps it open.

setpgrp() : ?
setpgrp() sets the process group. This is equivalent to setpgrp(0, 0) on BSD systems.

setpwent() : null
setpwent() resets and rewinds the pointer to the password file used by getpw() when getpw() is called with no arguments.

setservent(integer:1) : null
setservent(stayopen) resets and rewinds the pointer to the services file used by getserv().
The argument defines whether the file should be kept open between calls to getserv(); a nonzero value (the default) keeps it open.

setuid(integer) : ?
setuid(u) sets the user id of the current process to u. See the setuid(2) man page.

symlink(string, string) : ?
symlink(src, dest) makes a symbolic link dest that points to src.

umask(integer) : integer
umask(u) sets the umask of the process to u, an nine-bit encoding of the read, write, and execute permissions of user, group, and world access. See also chmod(). Each bit in the umask turns off that access, by default, for newly created files. The old value of the umask is returned.

wait(integer:-1, integer:0) : string
wait(pid, options) waits for a process given by pid to terminate or stop. The default pid value causes the program to wait for all the current process’ children. The options
parameter is an OR of the values 1 (return if no child has exited) and 2 (return for children that are stopped, not just for those that exit). The returned string represents the pid and the exit status as defined in this table:

<table>
<thead>
<tr>
<th>UNIX equivalent</th>
<th>example of returned string</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIFSTOPPED(status)</td>
<td>&quot;1234 stopped:SIGTSTP&quot;</td>
</tr>
<tr>
<td>WIFSIGNALLLED(status)</td>
<td>&quot;1234 terminated:SIGHUP&quot;</td>
</tr>
<tr>
<td>WIFEXITED(status)</td>
<td>&quot;1234 exit:1&quot;</td>
</tr>
<tr>
<td>WIFCORE(status)</td>
<td>&quot;1234 terminated:SIGSEGV:core&quot;</td>
</tr>
</tbody>
</table>

Currently the rusage facility is unimplemented.

## E.2 Microsoft Windows


### Partial support for POSIX

Windows supports `getpid()` but omits other process-related functions such as `getppid()`. On Windows `exec()` and `system()` may only launch Windows 32-bit .EXE binaries.

Windows Unicon supports the following signals in functions such as `kill()`: SIGABRT, SIGBREAK, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM.

Windows Unicon supports the `umask()` function, but ignores execute permission and treats user/group/world identically, using the most permissive access specified.

### Native user interface components

Windows Unicon supports limited access to platform-native user interface components and multimedia controls.

- `WinButton(w,s,x,y,wd,ht)` installs a pushbutton with label s on window w.
- `WinColorDialog(w, s)` allows the user to choose a color for a window’s context.
- `WinEditRegion(w, s, s2, x, y, wd, ht)` installs an edit box with label s.
- `WinFontDialog(w, s)` allows the user to choose a font for a window’s context.
- `WinMenuBar(w, L1, L2,...)` installs a set of top-level menus.
- `WinOpenDialog(w, s1, s2, i, s3, j, s4)` allows the user to choose a file to open.
- `WinPlayMedia(w, x[])` plays a multimedia resource.
- `WinSaveDialog(w, s1, s2, i, s3, j, s4)` allows the user to choose a file to save.
- `WinScrollBar(w, s, i1, i2, i3, x, y, wd, ht)` installs a scrollbar.
- `WinSelectDialog(w, s1, buttons)` allows the user to select from a set of choices.
Bibliography


Index

8-Queens problem, 36
abs, 363
absolute value, 363
abstract class, 206
access, 161
    functions, 83, 169
    structure, 157
    variable, 82
access time, set, 379
access, file, 68
acos, 363
action, statechart, 222
activate co-expression, 361
Active, 380
activity, statechart, 222
actor, 219
addition, 14
address space, 161
aggregation, 199
Alert, 380
Alexander, Bob, 46
aliasing, 348
alternation operator ( | ), 19, 50, 53, 272, 361, 430
alternation, repeated, 359
analysis
    post mortem, 160
    runtime, 160
AND operator, 53
any, 363
any(), 363
any(c), 45
APL, 3
apply operator, 54
apply(), 396
arc cosine, 363
arc sine, 363
arc tangent, 363
args, 363
args(p), 363
argument, 13
argument culling, 175
arithmetic, 14
arithmetic operator, 360
array, 31
ASCII, 16, 57, 337, 390, 430, 445, 447, 448, 450
ASCII, &ascii, 16, 349
asin, 363
assignment, 15, 39, 347, 360
    augmented, 359
    reversible, 360
    substring, 362
association, 179, 199, 219
associative memory, 30
atan, 363
atanh, 363
atomic types, 13
attribute
    class, 177
    GUI widget, 292
augmented assignment, 15
automatic instrumentation, 156
automatic storage management, 29
backtracking, 36, 41, 46, 450
backups, 252
bal, 363
bal(), 45, 51
balance string, 364
bang
    binary, 361
basename, 397
Bg, 380
binary data, 398, 412
binary operator, 15, 359
binomial coefficient, 419
BitString, 398
bitwise and, 369
bitwise or, 369
BNF, 411
Boehm, Barry, 1
Border, 473
bounded expressions, 19
bounded expressions, 362
bowdlerize, 419
break expression, 22, 355
built-in functions, 363
Button, 471
button, 290
ButtonGroup, 472
by, to-by step, 355
call depth, 351
call, procedure, 362
callback, 161
Cameron, Mary, 335
Cartesian coordinates, 198
case expression, 21, 355, 439
ceil(r), 420
center, 364
center(), 364
CGI, 227, 400
channel, 364
char, 364
character, 16, 42, 364
character set, 13, 15
event mask, 157
chart parser, 410
chdir, 364
chdir(), 67, 364
check boxes, 297
CheckBox, 475
CheckBoxGroup, 475
CheckBoxMenuItem, 479
chmod, 364
chmod(), 69, 364
chown(), 69
class
  abstract, 206
declaration, 179, 355
diagrams, 177
classname, 365
client, 65, 74, 263
client/server, 263
Clip, 380
clock tick, 166
Clone, 380
close, 365
close file, 365
close(), 66
closure, star(s), 409
co-expression, 55, 365
code reuse, 206
cofail, 365
collaboration diagram, 223
collate, 436
collect, 365
collect garbage, 365
Color, 380
color, 286
ColorValue, 380
column
  ODBC, 365
column number, 349
combinations, string, 436
command-line, 238
comment, 14, 215, 249, 432, 437, 445, 450, 453,
  458, 460
communication
tool, 168
comparison operator, 17
  string, 42
compile, 7, 27, 214, 389
compile time, 201
compiler, 24, 185, 378, 389
complement, cset, 359
complete(), 401
complex numbers, 401
Component, 468
compound expression, 362
concatenation, 361
condition, 17
condition variable, 365
conditional expression, 15
condvar, 365
conjunction &, 54, 361
constructor, 188, 281, 291, 346, 365, 425
class, 180, 199, 207, 356
dynamic record type, 97
record, 32, 184, 426
Container, 471
container, GUI class, 299
context switch
  lightweight, 162
close context-free grammar, 49
control structure, 13, 20, 41, 53, 56, 345, 349,
  354
  programmer defined, 362
crossover, 361
classifier, 471
conversion
type, see type, conversion
conversion, type, 358
convert
  base, 403
degrees to radians, 367
  radians to degrees, 375
to cset, 365
to integer, 369
to number, 372
to real, 375
to string, 377
copy, 365
copy(x), 34, 365
CopyArea, 381
cos, 365
cos(), 14
cosine, 365
Couple, 381
create, 55, 355
directory, 371
set, 376
critical section, 355
cross-reference, 451, 453
cset, 13, 15, 346, 365
event mask, 157
cset literal, 16, 42, 346
cset membership, 363
cset, universal &cset, 16, 349
cctime, 365
cctime(i), 365
current co-expression, 349
DAG, 438
database, 86, 87, 273, 346, 347, 365, 405, 448,
  457, 482
date, 349
date comparison, 403
days of the week, 408
dbcolumns, 365
dbdriver, 366
dbkeys, 366
dblimits, 366
DBM, 88, 367, 372, 390
dbproduct, 366
dbtables, 366
deeppcopy(), 35
default
  case branch, 355
  scanning parameters, 44
default parameters
    string scanning functions, 363
default value
      table, 30
default, parameter, 23
define symbols, 389
delay, 366
delay(i), 366
delete, 366
delete element, 366
delete()
    DBM database, 89
    from POP mail, 78
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>list, 31</td>
<td></td>
</tr>
<tr>
<td>set, 33</td>
<td></td>
</tr>
<tr>
<td>table, 31</td>
<td></td>
</tr>
<tr>
<td>deque, 31</td>
<td></td>
</tr>
<tr>
<td>dereference, 89, 359</td>
<td></td>
</tr>
<tr>
<td>design patterns, 206</td>
<td></td>
</tr>
<tr>
<td>detab, 367</td>
<td></td>
</tr>
<tr>
<td>Dialog, 467</td>
<td></td>
</tr>
<tr>
<td>dialog, 289</td>
<td></td>
</tr>
<tr>
<td>diff, 405, 447</td>
<td></td>
</tr>
<tr>
<td>difference (c1 -- c2), 16</td>
<td></td>
</tr>
<tr>
<td>difference (S1--S2), 33</td>
<td></td>
</tr>
<tr>
<td>digits, &amp;digits, 16</td>
<td></td>
</tr>
<tr>
<td>digits, cset &amp;digits, 349</td>
<td></td>
</tr>
<tr>
<td>directories, 65</td>
<td></td>
</tr>
<tr>
<td>directory, 67, 238, 364</td>
<td></td>
</tr>
<tr>
<td>create, 371</td>
<td></td>
</tr>
<tr>
<td>disk usage, 246</td>
<td></td>
</tr>
<tr>
<td>display, 66, 367</td>
<td></td>
</tr>
<tr>
<td>display(i,f), 367</td>
<td></td>
</tr>
<tr>
<td>division, 14, 420</td>
<td></td>
</tr>
<tr>
<td>do clause, 17</td>
<td></td>
</tr>
<tr>
<td>do, iteration, 355</td>
<td></td>
</tr>
<tr>
<td>DrawArc, 381</td>
<td></td>
</tr>
<tr>
<td>DrawCircle, 381</td>
<td></td>
</tr>
<tr>
<td>DrawCube, 381</td>
<td></td>
</tr>
<tr>
<td>DrawCurve, 381</td>
<td></td>
</tr>
<tr>
<td>DrawCylinder, 381</td>
<td></td>
</tr>
<tr>
<td>DrawDisk, 381</td>
<td></td>
</tr>
<tr>
<td>DrawImage, 382</td>
<td></td>
</tr>
<tr>
<td>DrawLine, 382</td>
<td></td>
</tr>
<tr>
<td>DrawPoint, 382</td>
<td></td>
</tr>
<tr>
<td>DrawPolygon, 382</td>
<td></td>
</tr>
<tr>
<td>DrawRectangle, 382</td>
<td></td>
</tr>
<tr>
<td>DrawSegment, 382</td>
<td></td>
</tr>
<tr>
<td>DrawSphere, 382</td>
<td></td>
</tr>
<tr>
<td>DrawString, 382</td>
<td></td>
</tr>
<tr>
<td>DrawTorus, 383</td>
<td></td>
</tr>
<tr>
<td>driver manager</td>
<td></td>
</tr>
<tr>
<td>ODBC, 92</td>
<td></td>
</tr>
<tr>
<td>DropDownList, 477</td>
<td></td>
</tr>
<tr>
<td>dtor, 367</td>
<td></td>
</tr>
<tr>
<td>dtor(r), 367</td>
<td></td>
</tr>
<tr>
<td>dynamic loading, 156, 162</td>
<td></td>
</tr>
<tr>
<td>e, 2.71... &amp;e, 14</td>
<td></td>
</tr>
<tr>
<td>E_MXevent, 167, 169</td>
<td></td>
</tr>
<tr>
<td>E_Opcode, 165</td>
<td></td>
</tr>
<tr>
<td>E_Pcall, 157</td>
<td></td>
</tr>
<tr>
<td>EBCDIC, 16, 390, 450, 451</td>
<td></td>
</tr>
<tr>
<td>EditList, 478</td>
<td></td>
</tr>
<tr>
<td>editor, 8, 183, 190, 447, 450, 458, 459, 462</td>
<td></td>
</tr>
<tr>
<td>elaboration, 219</td>
<td></td>
</tr>
<tr>
<td>elapsed time, 352</td>
<td></td>
</tr>
<tr>
<td>else, 355</td>
<td></td>
</tr>
<tr>
<td>empty list, 31, 362</td>
<td></td>
</tr>
<tr>
<td>encapsulation, 174</td>
<td></td>
</tr>
<tr>
<td>encryption, 446</td>
<td></td>
</tr>
<tr>
<td>end, 355</td>
<td></td>
</tr>
<tr>
<td>end-of-file, 16</td>
<td></td>
</tr>
<tr>
<td>entab, 367</td>
<td></td>
</tr>
<tr>
<td>environment variable, 368</td>
<td></td>
</tr>
<tr>
<td>CGI standard, 229</td>
<td></td>
</tr>
<tr>
<td>DateBaseYear, 404</td>
<td></td>
</tr>
<tr>
<td>EDITOR, 462</td>
<td></td>
</tr>
<tr>
<td>IPATH, 213, 395</td>
<td></td>
</tr>
<tr>
<td>IPL, 453</td>
<td></td>
</tr>
<tr>
<td>LPATH, 395, 452</td>
<td></td>
</tr>
<tr>
<td>PATH, 409</td>
<td></td>
</tr>
<tr>
<td>TERMCAP, 413</td>
<td></td>
</tr>
<tr>
<td>TRACE, 352</td>
<td></td>
</tr>
<tr>
<td>USER, 270</td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td></td>
</tr>
<tr>
<td>numeric =, 360</td>
<td></td>
</tr>
<tr>
<td>reference ===, 360</td>
<td></td>
</tr>
<tr>
<td>string ==, 360</td>
<td></td>
</tr>
<tr>
<td>equivalence</td>
<td></td>
</tr>
<tr>
<td>set, 434</td>
<td></td>
</tr>
<tr>
<td>EraseArea, 383</td>
<td></td>
</tr>
<tr>
<td>errno, keyword, 70</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td></td>
</tr>
<tr>
<td>convert to failure, 350</td>
<td></td>
</tr>
<tr>
<td>floating point, 70</td>
<td></td>
</tr>
<tr>
<td>message, &amp;errortext, 350</td>
<td></td>
</tr>
<tr>
<td>standard file &amp;errout, 350</td>
<td></td>
</tr>
<tr>
<td>standard, &amp;errout, 66</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

system, 393
errorclear, 367
errorclear(), 367
errortext, keyword, 67, 69
escape codes, 43
escape sequence, 406
eval(), 406
evdefs.icn, 163
Event, 383, 467
event, 156
  artificial, 167
categories, 165
code, 156, 157
mask, 157
pseudo, 167
report, 157
source, 163
value, 156, 157
virtual, 167
event code
  program execution, 350
event driven, 99
event driven programming, 168
event mask, 157
event stream
  dual, 163
event value
  program execution, 350
event(), 167
event, statechart, 222
eventmask, 367
eventmask(ce), 367
every, 20, 355
EvGet, 367
EvGet(), 164, 169
EvInit(), 163
evinit, 170
evinit library, 163
EvSend, 367
EvSend(x, y, C), 367
EvTerm(), 163
exclusive or, 369
execution
  control, 160
execution control, 167
execution model
  multitasking, 160
execution monitor
  Alamo, 158
exit, 367
exit(i), 367
exp, 367
exp(x), 14
exponential, exp(r), 367
exponentiation \^, 14
expression, 13
expression failure, 7, 11, 17, 21, 26, 30, 39, 50,
  53, 73, 345, 355
&fail, 351
factorial(n), 407
fail, 28, 51, 355
  co-expression, 56, 365
  expression, 18, 67
  system call, 69, 391
fallible expression, 18
Farber, Dave, 447
Feature(s), 439
features, 351
fetch, 367
fetch()
  DBM database, 89
  SQL, 94
fetch(d, k), 367
Fg, 383
field, record or class, 360
fieldnames, 368
fieldnames(r), 368
file, 65, 347, 412
  close, 365
  information, 67, 377
lock, 69, 368
  modified time, 68
permissions, 364
position, 379
redirection, 71
remove, 375
rename, 375
file ownership, 68
file size, 68, 377
filename completion, 46
FillArc, 383
FillCircle, 383
FillPolygon, 383
FillRectangle, 384
filtering, 157
find, 368
find string, 368
find(), 19, 44
findre(), 407
finite state machine, 221
floating point, 14
flock, 368
flock(f,s), 368
floor(r), 420
flush, 368
flush(f), 368
Font, 384
for loop, 20
fork(), 483
form feed, 459
fractions, 427
FreeColor, 384
function, 368
execution monitoring, 83
garbage collection, 30, 166, 349, 407
garbage collector, 365
Gaussian distribution, 408
generate elements, 358
generate operator 1x, 19, 358
generate sentences, 433
generator, 18, 21, 25, 28, 44, 46, 53, 56, 64, 345, 357, 361, 422, 426, 449, 452
function, 83
generator, random number, 426
genetic algorithms, 275
get, 368
getch, 368
getch, 368
getch, 368
getch, 368
getenv, 368
getpid(), 72
getserv(), 74
gettimeofday, 369
getuid(), 72
global, 212, 347, 356, 369
global variables, 24
globalnames, 369
goal-directed evaluation, 16
golden ratio 1.618..., &phi, 14
golden ratio, &phi, 351
GotoRC, 384
GotoXY, 384
grammar, 49
graph, 35, 189, 410
graphical user interface, 289
grep, 48, 407, 429, 451
gtime, 369
halt, 377
Hanoi, 26
hashing, 242
heap, 30
    EM separate from TP, 161
hexadecimal, 408, 410
host machine name, &host, 351
HTML, 78, 215, 227, 400, 410, 445, 453
HTTP, 78, 228
hyperbolic functions, 419
iand, 369
icom, 369
Icon, 473
Icon Program Library, 395
IconButton, 472
IdentityMatrix, 384
Idol, 174
if, 356
if statement, 17
if-then-else, 15
ifdef symbol, 389
Image, 473
image, 369
image(), 75
image(x), 369
immutable values, 13
import, 356
include, 389
independence
  of EM and TP, 161
index, subscript, 358
infinite loop, 357
infinity, 16
inheritance, 175, 187, 206
  multiple, 189
  semantics, 189
initial, 356
initial clause, 24
initially, 180, 356
input, 65, 412
  standard file &input, 351
input stream
  dual, 163
input, standard &input, 66
insert, 369
insert(), 29, 369, 481
  DBM database, 89
  list, 31
  set, 33
  table, 31
install, 7
instance, 37, 55, 187, 199, 281, 471
  class, 199, 207, 223, 348, 356
  record, 33
  superclass, 190
instances
  class object, 180
instrumentation, 156
integer, 13, 14, 345, 369
integer(x), 14
integrated development environment, 3
interface builder, 335
Internet, 65, 73, 263
intersection (c1**c2), 16
intersection (S1**S2), 33
inverse hyperbolic tangent, 363
invocable, 356
invocation, 361
  object method, 182
ior, 369
iplweb, 215, 453
IRC, 267
ishift, 369
istate, 369
iterator, 21
Ivib, 335
ixor, 369
Java, 3, 173, 178, 215
JavaDoc, 215
JavaScript, 236
julian, 403
kbhit, 370
kbhit(), 370
key, 370
  duplicate, 412
  ODBC, 366
key(), 248
  table, 31
key(x), 370
keyboard, 10, 66, 72, 298, 339, 368, 370, 390,
  415, 450, 471
keys, table, 30
keyword, 13, 370
keyword(), 370
KWIC, 453, 456
Label, 473
LaTeX, 447, 456
left, 370
left(), 370
length operator (*x), 16
letters, &letters, 16
letters, cset &letters, 351
lexical comparison, 18
library procedures
  monitor, 163
limitation \, 361
limiting an expression, 53
INDEX

line number, 351
link, 37, 212, 356, 395, 451, 460
    association instance, 223
    association instance, 199
    file system, 238, 394, 487
HTML, 400
    recursion, 238
    symbolic file, 487
link, file system, 68
Lisp, 3, 457
List, 477
list, 347, 370
    invocation, 54
    size (*L), 32
list concatenation
    L1 ||| L2, 32
list creation, 362
list data type, 31
list functions, 416
list(), 31
list(i, x), 370
literal, 13
literate programming, 4
literature, 457
load, 370
    C function, 370
    Unicon program, 370
load(), 370
loadfunc, 370
local, 347, 356, 370
local time, 365
local variables, 24
localnames, 370
lock, 370, 379
lock, file, 368
log, 370
log(x), 14
logarithm, 371
longest match, 418
loop, 12, 15
Lower, 384
lower case, 16, 48
lowercase, cset &lcase, 351

LU decomposition, 418
mail folder, read, 409
mail spool, 458
main task, 351
many, 371
many(), 371
many(c), 44
map, 371
map string, 371
map(), 59
match, 371
match string, 371
match(s), 45
matching functions, 44
matrix manipulation, 419
MatrixMode, 384
max, 371
max(), 14, 371
mean values, 420
member, 371
member(), 371
    set, 33
    table, 31
membernames, 371
memory addresses, 30
memory allocation, 30, 184
memory monitor, 166
memory regions
    separate, 161
memory use, 349, 352
Menu, 479
menu, 301
menu bar, 297
MenuBar, 478
MenuButton, 478
MenuComponent, 478
MenuEvent, 478
MenuSeparator, 479
messaging, 77
method, 179, 356
    combination, 191
    invocation, 360
overriding, 190
methodnames, 371
methods, 371
MIME, 397
min, 371
min(), 14, 371
mkdir, 371
mkdir(), 67, 371
modulo %, 14, 360
monitor
    anatomy, 168
    template, 168
monitor coordinators, 162
months of the year, 409
Morse code, 420, 458
move, 371
move(i), 44, 372
multidimensional array, 396
multiplication, 14
multiplicity, 200
MultMatrix, 384
mutable value, 346
mutable values, 29
mutex, 372
mutex(), 372
mutual evaluation, 361
MySQL, 92

n queens, 36
n-grams, 420
n-queens, 449, 461
name, 372
name mangling, 214
name space, 212
name(v), 372
natural log, &e, 350
natural log, log(x), 14
networking, 73
NewColor, 385
news, 458, 460
next, iteration, 356
nonnull test \x, 18, 23, 359
not, 54, 356
not equals, 11
now, 351
Nowlin, Jerry, 46
null test /x, 18, 23, 359
null value, 351
null value, &null, 13
numeric, 372
numeric comparison, 18
numeric operations, 14
object, 347
object-oriented programming, 4, 173, 174, 184
octal, 409
ODBC, 91, 96, 366, 372, 482
ODBC driver, 366
of, 357
open, 372
open file, 372
open(), 66
operations, 177
operators, 358
opmask, 373
oprec, 373
options(), 37
options, command-line, 421
OR operator, 53
ord, 373
ordinal value, 373
outline
    of an EM, 164
output, 65, 412
    standard file &output, 351
output, standard &output, 66
overlay, GUI class, 301
OverlayItem, 480
OverlaySet, 480
package, 207, 212, 357
PaletteChars, 385
PaletteColor, 385
PaletteKey, 385
palindrome, 437
Panel, 480
parallel evaluation, 57
parameter, 22, 30
parameter list, 11
parameter names, 373
parameters
wrong number, 22
paramnames, 373
parent, 373
parentheses, 362
parse, 49, 410, 460
password, 458
patch, 460
Pattern, 385
pattern matching, 15, 41, 49, 423, 449
patterns, design, 207
patterns, SNOBOL4, 423
peernames, 75
Pending, 385
Perl, 3, 429, 490
permissions, file, 364
permissions, file access, 65
permutations, 59
phi 1.618..., &phi, 14
phi, golden ratio &phi, 351
pi, 3.14... &pi, 14, 352
pipe, 71, 373
pipe(), 374
Pixel, 385
plural form, 424
pointer, 30, 34, 410
polling, 167
polymorphism, 29, 176
polynomials, 424
POP, 78
pop, 374
pop(), 31
from POP mail, 78
pop(L), 374
PopMatrix, 385
pos, 374
pos(i), 374
position, string, 43, 352
POSIX, 65
POSIX extensions, 483
PostgreSQL, 92
PostScript, 426, 447, 452, 461
power, exponent ^, 359
PowerBuilder, 3
predefined symbols, 389
preprocessor, 388
prime number, 407
printf(), 425
private, 178
proc, 374
proc(s,i), 374
procedure, 7, 13, 22, 357
procedure invocation, 399
process, 70
producer/consumer, 58
program, 7
program behavior, 158
program design, 176
program name, 352
program state, 80
programmer defined control structure, 362
protocol, 263
prototype, 1
proxy pattern, 207
pseudo events, 168
pseudo-tty, 250
public, 178
pull, 374
pull(L), 31, 374
push, 374
push(), 31, 374
PushMatrix, 386
PushRotate, 386
PushScale, 386
PushTranslate, 386
put, 374
put(), 31, 374
Python, 3
qualifier, 201
QueryPointer, 386
queue, 29, 31, 347, 368, 415
quota, 245
radial coordinates, 198
radian, 14, 198
radio buttons, 297
Raise, 386
random
  number generator, 426
  number seed, 60, 352
  operator, ?x, 359
randomize(), 426
rapid application development, 3
rational numbers, 427
read, 374
read(), 11, 66
read(f), 374
ReadImage, 386
reads, 374
reads(f,i), 374
ready, 375
ready(f,i), 375
real, 375
real number, 13, 346
real(x), 14
receive, 375
receive datagram, 375
receive(), 76
record, 32, 347, 357
record constructor, 365
recursion, 26, 34, 428, 451, 460
reduce(), 428
reference, 13, 22, 24, 30, 33, 178, 202, 207, 340
documentation, 215, 227
file, 66
HTML, 410
page, 456
reference comparison, 18
Refresh, 386
refresh, co-expression, 359
region sizes, 352
regular expression, 48, 407, 428
remainder, 15
remove, 375
remove directory, 375
remove file, 375
remove(s), 67
rename, 375
rename(), 67
repeat loop, 21, 357
repl, 375
replacement, string, 437
replicate list, 416
replicate string, 375
reserved word, 13, 354
result, 13
return, 357
reverse, 375
reverse(x), 16, 375
reversible assignment, 39, 360
reversible swap, 360
REXX, 3
right, 375
right(), 375
rmdir, 375
role, 201, 219
Roman numerals, 421
Rotate, 386
rtod, 375
Ruby, 3
run, 7
  loaded program, 81
  program under monitor, 160
run-time error, 14, 442
runerr, 375
runtime error, 375, 391
runtime system
  Unicon, 159
sampling, 169
Sapir-Whorf, 1
Scale, 386
scan string, 361
scanning
  environment, 43
  list, 416
scope, 213, 218, 348, 349
scope., 24
screen, 8, 289, 368, 462
index

scripting languages, 3, 227
ScrollBar, 474
sectioning operator, 18
seek, 375
seek(f,i), 375
segment(s,c), 433
select, 375
select(), 72, 251, 376
self, 179
semicolon insertion, 362
send, 376
send datagram, 376
send(), 76
sensor, 156
sentinel value, 16, 28
seq, 376
seq(), 55
sequence of results, 21
sequence, generate numeric, 376
sequences, 409
serial, 376
serial(x), 376
server, 65, 74, 264
set, 347, 376
set data type, 33
set(), 376
setenv, 376
setenv(), 376
SGML, 427, 464
shape equivalence, 405
shared address, 160
shell commands, 435
shift, 369
signal, 70, 376
Simula67, 174
sin, 376
sin(), 14
sine, 376
singleton, 207
size operator, 359
skeleton
  of an EM, 164
slice, 362

list L[i:j], 32
string s[i:j], 42
Smalltalk, 3, 173
SMTP, 78
SNOBOL4, 2, 41, 423
sort, 376, 414, 435
sort by field, 377, 435
sort source file procedures, 453
sort(x, i), 376
sortf, 376
sortt, 435
soundex, 435
source code, 8, 18, 86, 216, 280, 349, 395, 451
  line, 351
source code $line, 389
source file, 351
spawn, 377
spawn(), 377
special purpose monitors, 159
spiral model, 1
SQL, 90, 94, 273, 372, 482
  fetch, 367
sql, 377
sql(), 94, 377
sqrt, 377
sqrt(x), 14
square root, 377
stack, 26, 29, 31, 49, 347, 374, 440, 445
stat, 377
stat(f), 67, 377
state names, U.S., 436
statechart, 221
static, 36, 357, 377
static variables, 24
static, and class, 347
staticnames, 377
stop, 377
string, 13, 15, 346, 377, 436
  balance, 364
center, 364
closure, 409
comparison, 360
concatenation s1 || s2, 16
indexes 1 based, 41
indexes 1-based, 16
length (*s), 16
literal, 15
multi-line, 346
position, 43
scanning, 43, 432
subject, 43
subscript (s[i]), 16
string(x), 377
structure types, 22, 346, 443
subclass, 182, 187, 190, 210, 290, 310, 316, 467
subject string, 43
subject, string scanning, 352
SubMenu, 479
subscript, 362
subscript operator, 16, 18, 34, 89
subsection, 362
substring, 42
subtraction, 14
success, 17, 20, 358
superclass, 187, 360
operations, 190
suspend, 357
swap, 360
synchronous execution, see coroutine
syntax, 49
sys_errstr, 378
sys_errstr(i), 378
system, 377
system command, 378
system error, 393
system interface, 65
system(), 71, 247, 251

tab, 378
tab(i), 44, 378
tab, GUI class, 299
tab/match ( =s ), 359
TabItem, 480
Table, 480
table, 347, 378

length *T, 31
table data type, 30
table lookup, 30
table(x), 378
TableColumn, 479
TabSet, 480
talk, 267
tan, 378
tangent, 378
target program
Alamo, 158
task, 79
task switch, 162
Tcl, 3
TCP, 73
template
of an EM, 164
Texcoord, 387
TextButton, 471
TextField, 475
TextList, 476
Texture, 387
TextWidth, 387
then, 357
thread, 55, 358
time
of day, 369
since start, 352
time of day &clock, 349
time stamp, 349
to, generator, 19, 358
to-by, generator, 19
today &date, 349
tool communication, 168
Townsend, Gregg, 335
trace, 411
tracing, 27, 352, 449, 460
Translate, 387
trap, 378
trap signal, 378
trap(), 70
traverse, 210, 254
tree, 29, 35, 189, 211, 223, 238, 438
trigonometric function, 14
trim, 378
trim(s,c), 16
trim(s,c,i), 378
truncate, 378
truncate file, 378
trylock, 379
Turing machine, 465
two-way table, 438
type, 13, 379
conversion, 166
type conversion, 23, 358
type conversion, 14
type safe, 14
type(x), 379
UDP, 76
Ui, 8
umask, 69
UML, 176, 200
Uncouple, 387
undeclared variables, 24
undef, 389
union (c1 ++ c2), 16
union (S1++S2), 33
unlock, 379
until, 21, 358
upper case, 16, 353
upto, 379
upto(c), 44, 379
Usage(s), 439
use case, 218
user
Alamo, 158
input, 167, 168
interaction, 160
user interface, 289
utime, 379
uuencode, uudecode, 451
value, 13
value masks, 164
Variable, 24
variable, 13, 30, 179, 188, 335, 347, 356, 362,
370, 379
implicit, 179
no. of arguments, 54
variable(), 82
variable(s,c,i), 379
verse, 465
version, 353, 440
very high-level language, 2, 174
virtual machine
instruction, 165, 166
virtual monitor, 162
visibility
within class, 178
VisibleContainer, 471
visitor pattern, 210
Visual Basic, 3
VRML, 440
wait, 379
wait(), 72, 379
Wampler, Steve, 36
WAttrib, 387
WDefault, 387
WFlush, 387
where(f), 379
while, 12, 358
wild-card patterns, 441
WindowContents, 388
wrap, 431, 441, 448
write, 379
write(), 10, 66, 379
WriteImage, 388
writes, 379
writes(), 379
WSection, 388
WSync, 388
xcodes, 88, 442
XML, 236