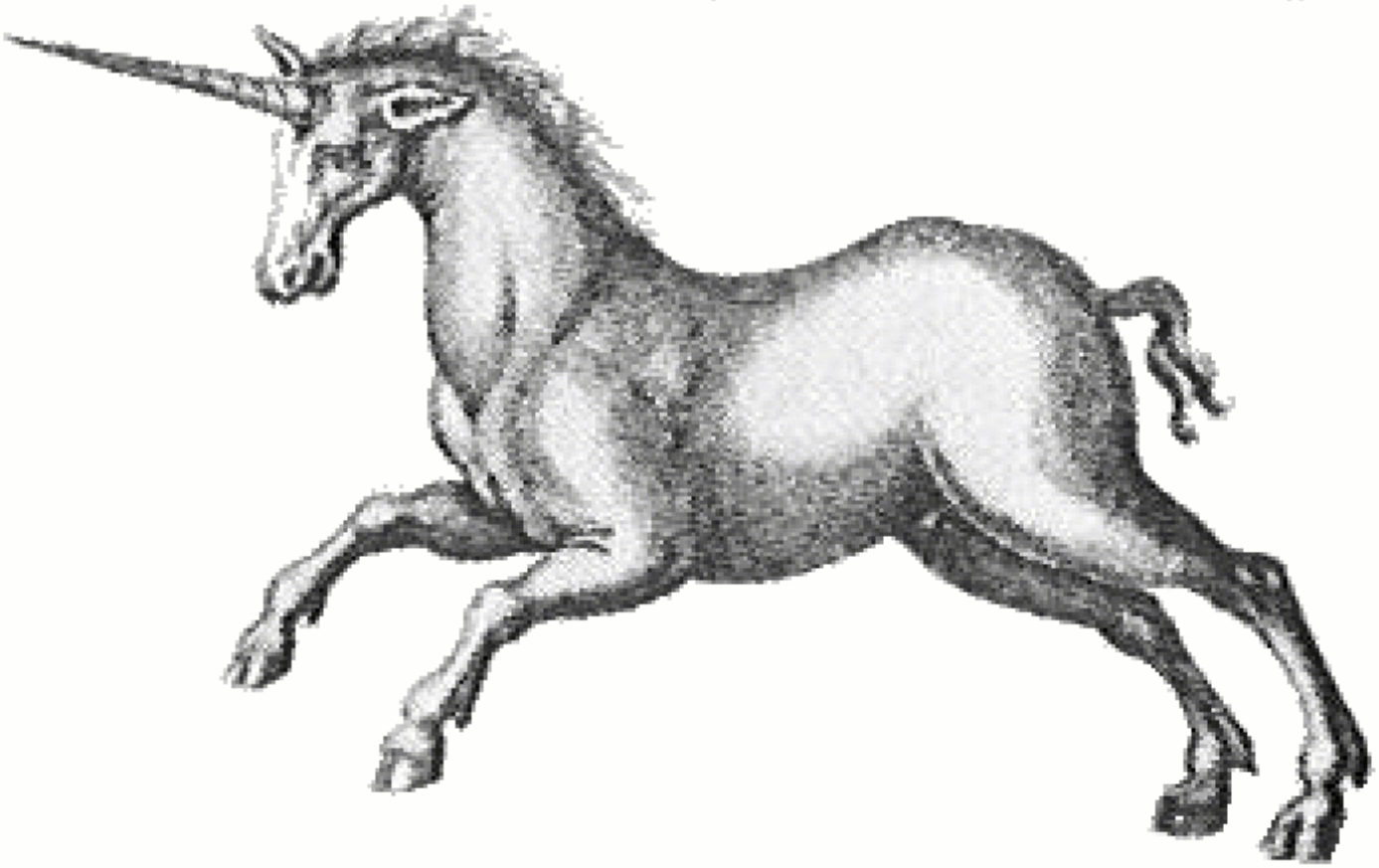

Programming with Unicon

Very high level object–oriented application and system programming

Jeffery, Mohamed, Pereda & Parlett

Programming with Unicon

Clinton Jeffery

Shamim Mohamed

Ray Pereda

Robert Parlett

This is a draft manuscript dated 9/28/2000. Send comments, errata and copy requests to jeffery@cs.unlv.edu

Contents

Preface

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 other 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, so it may (a) rhyme with "lexicon", (b) be The "un−Icon", (c) sound like "unicorn" in a plan or Bostonian accent, or (d) spell out the U and the N, suggestive of the "University of Nevada's descendant of Icon".

For current 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 and gives examples from new application areas to which Icon is well suited. For people new to Icon, this book is an exciting guide to a powerful language.

Organization of This Book

This book consists of four parts. The first part, Chapters 1−6, presents the core of the Unicon language that is derived primarily 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 7−11, 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 12−18 provide example programs that use Unicon in a wide range of application areas. Part four consists of essential reference material presented in several Appendixes.

Acknowledgements

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 most of the language and its program library over a period of many years. Icon contributors can be divided into epochs. In our epoch we have been inspired by contributions from Darren Merrill, Mary Cameron, Jon Lipp, Anthony Jones, Richard Hatch, Robert Shenk, Li Lin, David Rice, Joe van Meter, Federico Balbi, Todd Proebsting, and Steve Lumos.

The most impressive contributors are those whose influence on Icon has spanned across epochs, such as Steve Wampler, Bob Alexander, and Ken Walker. We revere you folks! Steve Wampler deserves extra thanks for serving as the technical reviewer for this book.

This manuscript received critical improvements and corrections from many additional technical reviewers, including Phillip Thomas, David A. Gamey, Craig S. Kaplan, David Feustel and Frank Lhota.

The authors wish to acknowledge generous support from the National Library of Medicine that enabled the development of the database facilities. This work was also supported in part by the National Science Foundation under grant CDA−9633299.

Clinton Jeffery

Shamim Mohamed

Ray Pereda

Robert Parlett

Introduction

Software development requires the ability to think about several dimensions simultaneously. For large programs, writing the actual instructions for the computer is not as difficult as figuring out the details of just exactly what the computer is supposed to do. After analyzing what is needed, program design brings together the data structures, algorithms, objects, and interactions that will accomplish the required tasks. Despite the importance of analysis and design, however, 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 essentially mathematical theorems, while programs are the 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 program implementation.

This book is a programmer's guide for an exciting programming language called Unicon. Programmers ranging from computer science researchers to hobbyists use Unicon. This book has something for the expert as well as the novice, as well as all programmers in between. You will find explanations of fundamental principles, hidden gems and language idioms, and advanced programming concepts and examples. You should study Unicon within the broader context of the field of software development, so we also cover a variety of practical software engineering fundamentals. We view writing a correct, working program as the central task of software engineering. This does not happen as an automatic consequence 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 demonstration for a subset of a software system. Prototypes are used to check software designs and user interfaces, to demonstrate key features to customers, or to prove the feasibility of a proposed solution. A prototype may help 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 generally done early in the software development process. It fits very naturally into the incremental, iterative *spiral model* of development proposed by Barry Boehm (1988). Figure I−1 shows the spiral model. This figure indicates time 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 even remotely implement the required functionality.

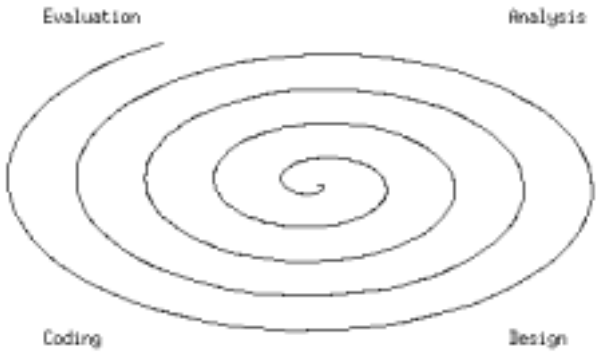

Figure I−1
The Spiral Model of Software Development

Tight spirals are better than loose spirals. The more powerful the prototyping tools used, the less time and money are expended in the early iterations of development. This savings 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 a technique to be used. This is OK, but a prototype that can be gradually enhanced until it becomes the final production system is more consistent with the spiral model.

Icon: a very high level language for applications

Icon is a very high−level language that originated at the University of Arizona. It is ideal for prototyping a wide range of applications. Icon benefits from the experience that its inventors gained from designing, implementing, and supporting a user community for an earlier very high−level language, SNOBOL4. Icon incorporates seminal research contributions to the field of programming languages. We love Icon not because it embodies neat research, but because it is more fun and easier to program in than other languages. Unlike most very high−level languages, which revel in cryptic syntax, Icon's syntax is not just more powerful, but in many cases more *readable* than the other programming languages of today. This gain in expressive power without sacrificing readability is an astonishing and addicting result of Icon's elegant design.

The current version of Arizona Icon is version 9.3.2. It is described in *The Icon Programming Language, 3rd edition* by Ralph and Madge Griswold (1996). The primary reference implementation of Icon is a virtual machine interpreter. The language evolved through many releases in the past two decades and is quite a bit more capable than it was originally. For example, Icon now includes portable high−level graphics facilities described in the book *Graphics Programming in Icon*, by Ralph Griswold, Clinton Jeffery, and Gregg Townsend (1998). Icon is mature, and its designers do not envision major additions to the language at this point. Over the past decade the emphasis has been on increasing Icon's suitability for a wide range of applications. At the University of Arizona, examples of this work have included implementations of Icon

that generate C (which is compiled to native machine code for speed), and Java virtual machine code (for portability).

Enter Unicon: More Icon than Icon

The name Unicon refers to the descendant of Icon described in this book and distributed by the University of Nevada. Unicon is Icon with the authors' additions: elegant, portable, and platform−independent facilities that have become ubiquitous in 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 didn't feel free to call it version 10 of the Icon language, since it is not produced or endorsed by the Icon Project at the University of Arizona. They are not responsible for Unicon's bugs, nor does it make their standard release of Icon out of date.

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 99.9 percent backward compatible (this measurement is figurative, not literal) with Icon, dropping 0.1 percent 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. One kind of competition comes from mainstream systems programming languages such as C, C++, and Java. Like the assembler languages that comprised the mainstream before them, these languages are ideal tools for writing all sorts of programs, so long as vast amounts of programmer time are available. The computing culture has long understood that throwing more programmers at a big project is a poor solution, and programmers are getting more expensive while computing resources continue to become cheaper. These pressures slowly but inexorably lead to the use of higher−level languages and the development of better design tools and development methods. Such human changes are incredibly slow compared to technological changes, but they are visibly taking place nevertheless. Today, many of the most productive programmers are using extra CPU cycles and megabytes of RAM to make it several times faster to develop useful programs.

There is a subcategory of mainstream languages, marketed as *rapid application development* (RAD) languages, whose stated goals seem to address this phenomenon. Languages such as Visual Basic, Delphi, and 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 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, putting it in the category of "very high level languages." This category includes Lisp, APL, Smalltalk, REXX, Perl, Tcl, and Python; there are many others. Very high−level languages may be further subdivided into scripting languages and applications languages. Scripting languages are generally designed to 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 can be viewed as just one prominent subcategory of very high−level languages.

Icon is an applications language; its SNOBOL roots are in text processing and linguistics. For many application areas, Icon is the best programming language currently available. Icon programs are written faster, in less code, than conventional languages such as C or Java. Icon programs tend to be more readable and maintainable than similar programs written in other very high−level languages. This makes Icon particularly well−suited to the aims of *literate programming*; for example, Icon was used to implement the popular literate programming tool noweb. Literate programming is the practice of writing programs and their supporting textual description together in a single document. This book includes numerous examples that illustrate the range of tasks for which Icon is well suited, and these examples are the best evidence we can provide in support of our argument.

Consider using Icon when one or more of the following conditions are true. The more conditions that are true, the more likely you will benefit from Icon.

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

Icon is not the last word in programming. You probably should not use Icon 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.

- The program must utilize substantial libraries written in lower−level languages such as C, C++, or FORTRAN.

Programming languages play a key role in the software development enterprise. Icon is an exciting very high−level language with a unique design that reduces programming time without sacrificing readability. The Unicon language extends Icon for use in larger programs that may be interconnected using the Internet, such as object−oriented programming and database capabilities. This book will show you how Icon makes complex programming tasks easier. Many examples from a wide range of application areas serve to demonstrate how to apply and combine language constructs to solve real−world problems.

It is time to move past the introductions. Prepare to be spoiled with one of the most hassle−free programming languages you'll ever use. You will experience the same feelings that early Italian merchants 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 Icon

Chapter 1: Programs and Expressions

At Unicon's heart are the core language features it share with Icon. This chapter presents many of the key features of Icon, starting with elements that Icon has in common with other popular languages. Detailed instructions show you how to compile and run programs. Soon the examples start to focus on important ways in which Icon is different from other languages. These differences are more than skin deep. If you dig deeply, you can find dozens of details where Icon provides just the right blend of simplicity, flexibility, and power.

When you finish this chapter, you will know how to

- edit, compile, and execute Icon 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

Your First Icon Program

This section presents the nuts and bolts of writing and running an Icon program. The information in this section is essential in order 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 E for details on installing Unicon from the accompanying CD−ROM.) 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 various combinations of commands that either use hardware to obtain or present information to users, or else perform internal calculations that transform or manipulate information into a more useful form. To program a computer you write a document containing instructions that the computer will carry out in some order. In Icon a list of instructions is called a *procedure*, and a *program* is just a collection of one or more related procedures. Program files may be composed using any text editor. For the purposes of demonstration this section describes how to use Wi, the program editor and integrated development tool that comes with Unicon.

It is time to begin. Fire up Wi by launching the icon labeled "Windows Unicon," and type:

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

Subject to font variations, 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 Wi 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 Wi uses to display source code is selectable from the Options menu.

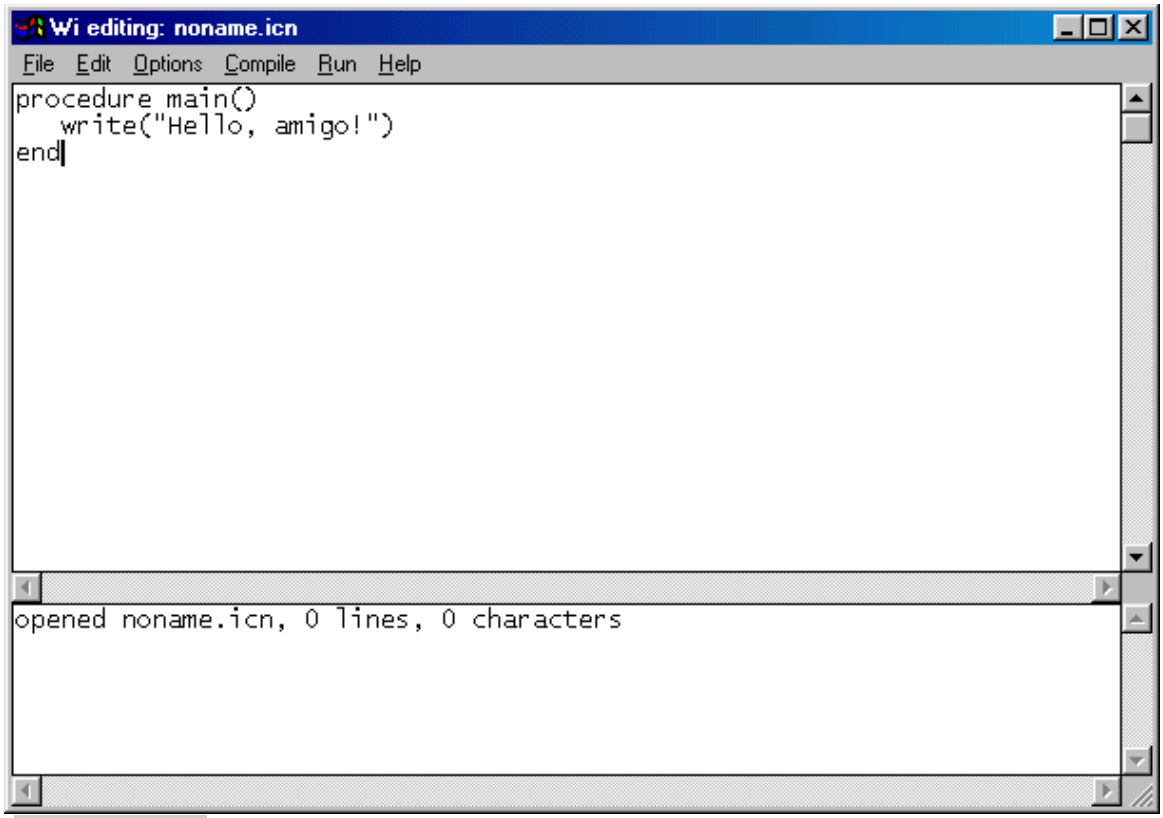

Figure 1–1:
Writing an Icon program using the Wi program.

The format of the instructions deserves an explanation: In Icon, lists of instructions are called *procedures.* Procedures begin with the word procedure and end with the word `end`. Each procedure is given a name, so that after writing a list of instructions you may refer to it by name without writing out the whole list again. The `write()` instruction is just such a procedure, only it is already written for you and comes built in to the Icon language. When you issue a `write()` instruction, you must tell the computer what to write. The directions a procedure uses in carrying out its instructions are given inside the parentheses following that procedure call; in this case, `"Hello, amigo!"` is to be written.

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 Wi. For instance, if you select **Run¢Run**, Wi will do the following things for you.

1. Save the program in a file on disk. All Icon programs end in `.icn`; you could name it anything you wished, using the **File¢SaveAs?** command.

2. Compile the Icon 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. Wi 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.

Somewhere in this discussion the word `end` needs to be introduced. The word `end` marks the completion of the list of instructions that make up a procedure. 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 value 19 away, and go on.

Add the following line, and run it:

```
   write("7 + 12")
```

This illustrates what the quotes are for. Quoted text is taken literally, but without quotes, the computer tries to simplify (that is, do some arithmetic), which might be difficult if the material in question is not an equation!

```
   write(hey you)
```

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

```
   write(7 + "12")
```

Whatever is in quotes is a single entity; if it happens to be a number, adding it to another number makes perfect sense. 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
```

After running the program to see what it does, you should notice the following: The procedure named read() is used to get what the user types; you can tell it's a procedure call because of the parentheses that follow it. 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 1 value on a line, separated by commas. 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 in effect 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. There is one more point hidden in the program above. When you run the program, the `"this line says"` part does not get printed until after you type the line for `read()`. The `write()` instruction must have all of its directions (the values inside the parentheses) before it can go about its business. The term for directions one gives to a procedure is *parameter*; 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").
```

Run the following program:

```
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.

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. All of this explains the purpose of the `while` instruction. It is a "loop" that executes repeatedly the instruction that follows it until it fails. 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 Icon 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 context 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 Icon a unique experience.

Expressions and Types

Each procedure in an Icon program is a sequence of *expressions*. All expressions are instructions for obtaining values of some type, such as a number or a word; some expressions also cause side effects, such as accessing a hardware device to read or write data. Simple expressions directly name 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 name a value stored in memory. When the program runs, they do not introduce 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 Icon 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. Values of other types such as lists must be constructed during execution. Later chapters describe structure types that organize collections of values, and system types for files, windows, network connections and other system entities.

After literals, references to *variables* are the next simplest form of expression. Variables are 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.

Certain words may not be used as variable names because they have a special meaning in the language. Some of these *reserved words* are `if`, `while`, and so on. Certain other special variables called *keywords* start with the ampersand character (`&`) and denote special values. All variables are initialized to the null value represented by the keyword `&null`. Some of the other keywords are `&date`, `&time`, and so on. We will describe reserved words and keywords when the corresponding language features are presented. Complete lists of reserved words and keywords are given in Appendix A.

Unlike many languages where every variable must be declared along with its data type, in Icon variables do not have to be declared at all, and any variable can hold any type of value. However, Icon will not allow you to mix incompatible types in an expression. Icon 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.

Numeric Computation

Icon supports the usual arithmetic operators on two data types: integer and real. The integer type is a signed whole number 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.

As a general rule in Icon, 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 allows 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* is available in keyword `&pi`, the Golden Ratio (1.618...) is available in `&phi`, and *e* 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; they are library procedures in Icon and built−in functions in Unicon.

Listing 1−1 shows a simple Icon program that illustrates the use of variables in a numeric computation. The three lines at the beginning are comments for the human reader. The compiler ignores them. Comments begin with the # character and extend to the end of the line on which they appear. The `link numbers` declaration brings in numeric procedures such as `min()` from the Icon Program Library; it is not really needed here under Unicon, but illustrates the general method for accessing code libraries.

Listing 1−1
Mystery program

```
#
# a mystery program
#
link numbers
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 arithmetic modulo `%` operator is an important part of this program: `i % j` is the remainder when `i` is divided by `j`.

The while loop in this program uses a reserved word `do` followed by an expression (really a compound expression enclosed in curly braces). The expression following the `do` is executed once each time the expression that controls the `while` (`r > 0`) succeeds. Inside the `while` loop, a conditional `if–then–else` expression is used to select one of two possible actions.

The names of the variables in this example are obscure, and there are no human readable comments in it, only the one telling you it's a "mystery program'!" 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, Icon also offers the *augmented assignment* operators. For instance, 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 `i` *op*`:= expr` operator means the same as `i := i` *op* `expr`.

Strings and Csets

The non–numeric atomic types available in Icon are character sequences (strings) and character sets (csets). 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 in Icon; a string (or cset) consisting of a single character is used instead.

Current implementations of Icon 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 `s1 || s2` expression produces a new string consisting of the characters in `s1` followed by those in `s2`. The subscript operator `s[i]` produces a one–letter substring of `s` at the `i`th position. Indexes are counted starting from position 1. If `i` is negative, it is relative to the end of the string.

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

Icon originated in the domain of string processing, and has many sophisticated features for manipulating strings and performing pattern matching. This section presents simple operations. More advanced operations using strings and csets are given in Chapter 4. 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.

Goal−directed Evaluation

So far, the simple examples of how expressions are evaluated by Icon have included nothing out of the ordinary. 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:

```
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. Icon provides a much nicer way to write this type of code. In Icon, 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 have to save the value in the intermediate variable `i`. If you would like to say:

```
process(read())
```

then your wishes are answered by Icon. You can indeed write your program like this. The expression `read()` tries to produce a value by reading the input. When it is successful, `process()` is called with the value; but when `read()` cannot get any more values, that is, at the end of the file, it *fails*. This failure propagates up and `process()` is not called either. And 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

```
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:

```
if i > j 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. In Icon you can write conditions like

```
if 3 > i > 7 then ...
```

that appear routinely in math classes. Other programming languages only dream about being this elegant. First, Icon 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.)

Fallible Expressions

Because some expressions in Icon 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 Icon 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 main 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 `read()` function is illustrative of a large number of built−in functions that can fail. A call to `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. 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 `\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.

`<`	`<=`	`>`	`>=`	`=`	`~=`	numeric comparison operators
`<<`	`<<=`	`>>`	`>>=`	`==`	`~==`	lexical (alphabetic) comparison
`===`		`~===`				reference comparison

Generators

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

```
find("lu", "Honolulu")
```

In most languages, this would return one of the possible substring matches, usually the first position at which the substring is found. In Icon, this expression is a generator, and is capable of producing *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 < find("or", "horror")
```

The first value produced by `find()` is 2, which causes the `>` operation to fail. Icon then resumes the call to `find()`, which produces a 5 as its next value, and the expression succeeds. Furthermore, the value of the expression is the first position of the substring that is greater than 3.

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

```
expr1 | expr2
```

is a generator that produces its left−hand side followed by its right−hand side, if needed by the surrounding expression. This can be used to perform many computations quite compactly. For example,

```
x = (3 | 5)
```

checks to see if the value of `x` is either 3 or 5. This can be extended to more complex expressions:

```
(x | y) = (3 | 5)
```

This checks to see if either of `x` or `y` has the value 3 or 5. It is the Icon equivalent of C's

```
(x == 3) || (x == 5) || (y == 3) || (y == 5)
```

In understanding any piece of Icon code you come across, it helps if you identify the generators, if there are any. In addition to the alternation operator `|` and the function `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`, and 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 all 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); to a set, table, list, or record, it produces all the members of the structure.

Iteration and Control Structures

You have already seen two control structures in Icon: the `if` and the `while` loop, which test for success. The control structure `every` is a mechanism for processing the entire sequence of values produced by a generator. The expression

```
every expr1 do expr2
```

evaluates *`expr2`* for each result generated by *`expr1`*. This is a loop, and it looks similar enough to a `while` loop to confuse people at first. The difference is that a `while` loop re−evaluates *`expr1`* from scratch after each iteration of *`expr2`*, while `every` *resumes* *`expr1`* for an additional result right where it left off the last time through the loop. Using a generator to control a `while` loop makes no sense; the generator will just be restarted each iteration, which may give you an infinite loop. Similarly, *not* using a generator to control an `every` loop also makes no sense; if *`expr1`* is not a generator the loop body will execute at most one time, similar to an `if` expression.

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 *`expr2`*. In many languages these are called "for" loops. A for loop in Icon is written like this:

```
every i := 1 to 10 do write(i)
```

Of course, the point of `every` and `to` is not that you can use them to implement a for loop. Icon’s generators are a lot more flexible. In fact, the for loop above looks pretty clumsy when you compare it with the equivalent

```
every write(1 to 10)
```

Icon’s `every` keyword generalizes the concept of *iterators* found in languages such as Clu and C++. Iterators are special constructs added to walk through a collection of data. Instead of being a special feature, in Icon 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 in Icon produce an infinite sequence of results, as long as you keep resuming them for more.

Here is another example of how Icon’s `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. An example in the next section shows how you can generalize this to work with all the squares.

The `if`, `while`, and `every` expressions are the kingpins of Icon’s 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 ...` expressions. The `case` expression replaces these chains of `if` expressions with the syntax:

```
case expr of {
branch1: expr1
branch2: expr2
...
default: expr_i
}
```

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:

`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.

Procedures

Procedures are a basic building block in most languages, including Icon. Like C, an Icon program is organized as a collection of procedures and execution starts from a procedure named `main()`. Here is an example of an ordinary procedure. This one computes a simple polynomial, ax^2+bx+c.

```
procedure poly(x,a,b,c)
   return a * x^2 + b * x + c
end
```

Parameters

Parameters to the procedure are passed by value except for structured data types, which are passed by reference. This means that if 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 passed by reference. The procedure being called gets a handle to the original value, so that 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. 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. Null values are used to tell a procedure to use a default value for a parameter, if one is defined.

Now it is appropriate to describe more properly the unary operators `\` and `/`. These test the expression against the null value. The expression */expr* succeeds and returns *expr* if the value is the null value. This can be used to assign default values to procedure arguments: Icon will use the null value to any arguments that the caller does not specify. Here's an example:

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

If this procedure is called as `succ(10)`, Icon will assign the null value to the missing second parameter, `i`. The forward slash operator then tests `x` 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 preceding example of procedure `succ()` shows one way to specify a default value for a parameter. Icon's 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 Icon's built−in repertoire is implicit type conversion. Arguments to built−in functions and operators are converted as needed.

Defaulting and type conversion are so common in Unicon and so valuable especially in library routines that they have their own syntax.

```
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

The variable names used as procedure parameters are fundamentally different from the names of procedures. Parameter names have a *scope* that is limited to the body of a single procedure, while procedure names are visible through the entire program.

Parameters and procedures are special forms of the two basic types of variables in Icon: local variables and global variables.

Global variables, including procedure names, have a scope that consists of the entire program and store their value at a single location in memory for the entire execution of the program. There are two kinds of local variables; both kinds 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 just global variables whose names are defined 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 in Icon 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:

```
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:

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

Although you do not have to declare local variables in Icon, for any larger program and especially for any library procedures or multi−person projects it is a mistake to not declare all local variables. The reason it is a mistake is that if 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 you avoid this problem, the `−u` command line option to the Icon compiler causes undeclared local variables to produce a compilation error message.

An Icon procedure body can begin with an `initial` clause, which will execute only the first time the procedure is called. The `initial` clause is primarily used to initialize static variables. As an example, the following procedure returns the next number in the Fibonacci number sequence each time it is called, using static variables to remember previous results in the sequence between calls.

```
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! A generator is written by using `suspend` instead of `return`; if another value is required, 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.

```
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:

```
   every munge(squares())
```

> **Warning**
>
> This is an infinite loop! (Do you know why?)

The `fail` expression makes the procedure fail. Control goes back to the calling procedure without returning a value, and the procedure ceases to exist; it cannot be resumed. A procedure also fails implicitly if control flows off the end of the procedure's body.

Here is a procedure that will produce all the non−blank characters of a string, but bailing out if the character # is reached:

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

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 were presented with a problem. In a temple in some remote place, 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 complete their task, the universe will come to an end.

How can you move the *n* smallest disks? Well, if *n* is 1, it is simple; just move it. Since it's the smallest, we will not be violating 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 `hanoi(n)` that computes this algorithm. The first part is simple: if you have one disk, just move it.

```
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
```

Now move the *n−1* disks from `needle1` to `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}.

```
   hanoi(n-1, needle1, other)
   write("Move disk from ", needle1, " to ", needle2)
   hanoi(n-1, other, needle2)
```

And that's it! You're done. Listing 1−2 contains the complete program for you to try:

Listing 1−2
Towers of Hanoi

```
procedure main()
   write("How many disks are on the towers of Hanoi?")
   hanoi(read())
end

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

You might want to try this out with tracing turned on to see how it works. To enable tracing, compile your program with a `-t` option, or assign the keyword &trace a non−zero number giving the depth of calls to trace. Setting &trace to a value of −1 will turn on tracing to an effectively infinite depth.

Incidentally, to move *n* disks, $2^n - 1$ individual disk movements will be required. If the monks can move one disk a second, it will take them $2^{64} - 1$ seconds, or about 60 trillion years. Current cosmology has the age of the universe to be around 50 billion years old, so we probably don't have to worry about the monks finishing their task!

Summary

In this chapter you have learned:

- Icon is an expression−based language organized as a set of procedures starting from a procedure called main().
- Icon has four atomic types: arbitrary precision integers, real numbers, arbitrary length strings of characters, and character sets.
- There is no Boolean concept; 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. Icon features extensive argument defaults and automatic type coercion throughout its built−in function and operator repertoire.
- Icon has two primary 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* in Icon. For example, all operations that manipulate numbers and strings compute new values, rather than modify existing values. This chapter presents Icon's 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 Icon 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 Icon'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, Icon'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 Icon performs automatic storage management. 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. Icon 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.

Icon has no pointer types. Instead, all structure values implicitly use pointer semantics. A *reference* is a pointer for which type information is maintained and type safety is strictly enforced. In Icon, all structure values are references to structure data that is allocated elsewhere, in a memory region known as the heap. If you are a C programmer, 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 `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 `copy()`. To copy a "deep" structure (lists of lists, tables of records, etc.) you can use the procedure `deepcopy()` that is given as an example later in this chapter.

Tables

Tables are unordered collections of values that are accessed using associated *keys*. They are Icon'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 `table()` function creates a new empty table. For example, the lines

```
T := table()
T["hello"] := "goodbye"
```

create a new table, and associate the key `"hello"` with the value `"goodbye"`. The `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, `write(T["goodbye"])` would write an empty line, since `write()` treats a null argument the same as an empty string, and `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"])` would write out `"goodbye"`.

Subscripts are the primary operation on tables, but there are several other useful operations. The `insert(T, k1, x1, k2, x2, ...)` function adds new key−value pairs to `T`. The `delete(T, k1, k2, ...)` 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`, `*T` is the number of key−value pairs in the table. The unary `!` operator generates elements from any collection; for a table `T`, `!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.

The `member(T, k)` function succeeds if `k` is a key in `T` and fails otherwise. The `key(T)` function generates the keys with which values have been stored. Here is a code fragment to print a word count of the input (assuming the `getword()` procedure generates words of interest):

```
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.

Lists

Lists are ubiquitous dynamically sized ordered collections of values. They are accessed using subscripts, with indexes starting at 1. You can also modify lists by inserting or removing 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 lists'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:

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

Lists are dynamic. `insert(L, i, x)` inserts `x` at position `i`, and `delete(L, i)` deletes the element at position `i`. In addition, 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. The previous list could have been constructed one element at a time with the following code. The expression `[]` creates an empty list; it is equivalent to calling `list()` with no 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. Both L[2][3] and the equivalent to L[2,3] yield the value 6 in this example.

Lists also support several common operators. The unary operator `*L` produces the size of list `L`. The unary 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 lists with tables, lists are like tables with really boring keys, the positive integers starting from 1. The `member(L, k)` function succeeds only if `0 < integer(k) < *L`. The `key(L)` function 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`.

Records

A record is a fixed−sized, ordered collection of values whose elements are usually 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.

```
record complex(re, im)
```

Record *instances* are created using a constructor procedure with the name of the record type. On such instances, fields are accessed by name using a dot notation or string subscript, or by integer index subscript. The upshot of all this is that you can use records as records, or as special tables or lists with a constant size and fixed set of keys.

Functions like `insert()`, or `push()` are not supported, since they change the size of the structure that they modify. The following example illustrates some 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")
```

Records are closely related to classes and objects, which are discussed in Chapter 7. They can be considered to be an optimized special case of objects that have no methods. Records are important because they are Icon’s only means of introducing new data types. 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 7.

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(L)` creates a set containing the elements from list `L`. 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 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 Icon value, it can contain a reference to itself. This is one of several differences between Icon sets, which are mutable structures, and mathematical sets. Another difference is that Icon 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.

```
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 can be considered 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.

Using Structures

The elements of structures can be other structures, allowing you to organize your information in whatever way best fits your application. In Icon building complex structures such as a table of lists, or a list of records that contain sets, requires no special trickery or new syntax. A few 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 our 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 any of the structure types. If you have a multi−level structure, you need to call copy for each substructure if you wish the new structure to not have any pointers into the old structure. This is a natural task for a recursive function.

```
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))
      }
   default: return x
   }
```

```
   return y
end
```

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:

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

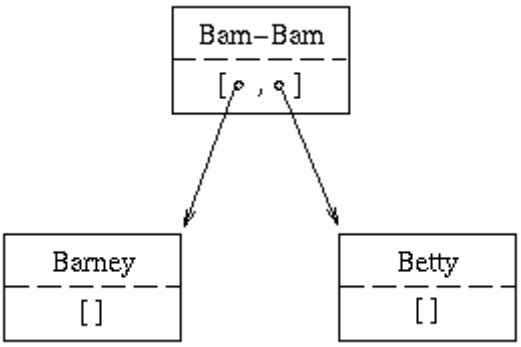

Figure 2−1:
A Record Containing a List of Two Records

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 | !n.links do {
      if not member(relatives, i.name) then {
         write(i.name)
         insert(relatives, i.name)
         print_relatives(i)
         }
      }
end
```

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 they serve here? For a proper tree structure, they are not needed at all, but for non−trees they are very important! One defect of this procedure is that there is no way to reset the static variable and call `print_variables()` starting from scratch. How would you go about removing this defect?

The *n*−Queens Example

The 8−Queens problem is the classic backtracking problem. The problem 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` serves to store 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.) Both the problem size `n` and the array variable 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
```

The program starts by handling command−line arguments. For every Icon program, `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: an option that specifies what size of board is to be used. Thus `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.

```
   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:

```
   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.

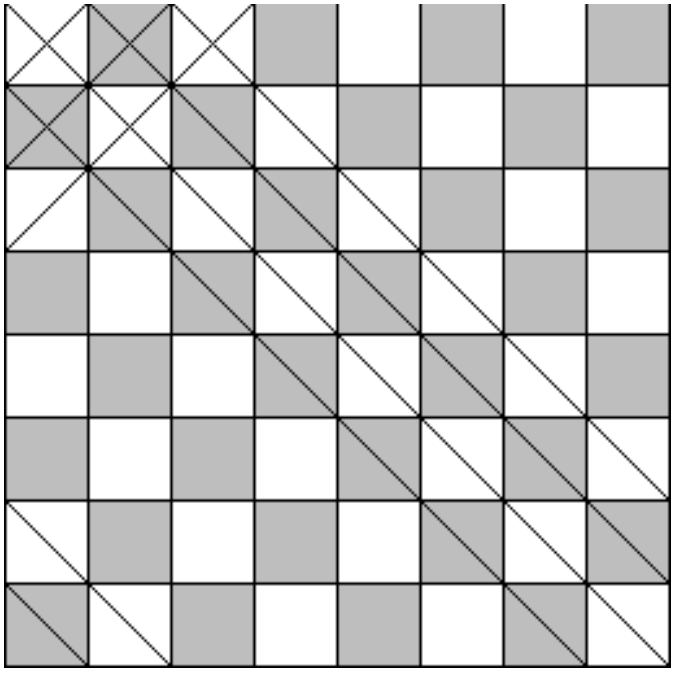

Figure 2−2:
Up and Down Diagonals in the n−Queens Problem

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

1. `rows[r]` is zero, that is, no other column has a queen in row `r`,

2. `up[n+r−c]` is 0, that is, there is not already a queen in the "up" diagonal, and

3. `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:

```
   every 0 = rows[r := 1 to n] = up[n+r−c] = down[r+c−1] &
      rows[r] <− up[n+r−c] <− down[r+c−1] <− 1 do {
```

Note that for assignment, instead of `:=` this expression uses the *reversible assignment* operator `<−`. This assigns a value just like the conventional assignment, but it remembers the old value; if it is ever resumed, it restores the old value of the variable and fails. This means that between iterations, the appropriate entries in the `row`, `up`, and `down` arrays will be reinitialized.

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:

```
      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.

```
#
# 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)
   write("  ", border)
   every line[4*(!solution − 1) + 3] <− "Q" do {
      write("  ", line)
      write("  ", border)
      }
   write()
end
```

Summary

Icon’s structures are better than sliced bread. They are the foundations of complex algorithms and data structures. They are every computer scientists’ fully buzzword−compliant best friends: polymorphic, heterogeneous, implicitly referenced, cycle−capable, dynamically represented, and automatically reclaimed. Later on we will see that 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

Besides expression evaluation, Icon offers compelling features to reduce the effort required to write complex programs. Its ancestor is SNOBOL4, the grandfather of all string processing languages, and from it Icon inherits 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

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 last position of the string, and negative indices count from the right side of the string:

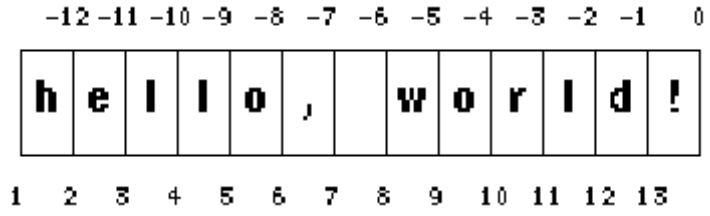

Figure 3−1:
Positive and Negative String Indices

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] := " puny "
s[13:18] := "earthlings"
```

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:

```
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 `==`.

```
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.

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:

```
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.

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 3−1
Escape Codes and Characters

Code	Character	Code	Character	Code	Character	Code	Character
`\b`	backspace	`\d`	delete	`\e`	escape	`\f`	form feed
`\l`	line feed	`\n`	newline	`\r`	carriage return	`\t`	tab
`\v`	vertical tab	`\'`	quote	`\"`	double quote	`\\`	backslash
`\`*`ooo*	octal	`\x`*`hh`*	hexadecimal	`\^`*`x`*	Control−*x*		

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

`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 will usually have a few *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 alternate character of the string in variable `text`.

Another function is `tab()`, 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 is to look 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:

```
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:

```
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 Icon to resume the first `tab()`, which resets the position to the beginning of the string and fails. Icon then evaluates the expression `tab(many(&digits))` which succeeds (matching the string `"0"`); therefore the entire expression succeeds and leaves `&pos` at 2.

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

```
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 := 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], "\t")
      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 itself takes an 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, that line converts the line just read in to all lowercase.

Regular Expressions

The Icon Program Library (included with the distribution) provides regular expression matching functions. To use it, include 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("cant 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 Icon program has access to the operation at a finer grain, more complex operations are possible, this search−and−replace is just an example.

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. They are 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 Icon is described in Chapter 16. 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 −> S S
   | (S)
   | ()
```

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 the rule, and *terminals* like (

and). A series of applications of rules is called a derivation. One non−terminal is special and is called the *start symbol*; a string is matched by the grammar if there is a sequence of derivations from the start symbol that leads to the string. By convention the start symbol will be the one mentioned first 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:

```
S −> (S) −> (SS) −> (()S) −> (()(S)) −>
     (()(SS)) −> (()(()S)) −> (()(()()))
```

Parsing

Icon programs can parse grammars in a very natural way using matching functions. A production

```
A −> B a D
   | C E b
```

can be mapped to this matching function:

```
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, Icon will backtrack 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 Icon 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 would be a very expensive way to do it! Notice that Icon's expression evaluation will basically 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 Icon versions of the standard industrial−strength lexical analyzer and parser generation tools, lex and yacc.

Doing It Better

Many grammars can be parsed much more efficiently by using the well−known techniques − consult a book on compilers for details. Here we present one way of parsing a grammar using some of the built−in functions Icon has. 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 Icon 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(tab(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 `term()` and `expr()` have the same structure. The `term()` 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 `factor()` uses string concatenation instead of conjunction, so that it can return the matched substring.

Summary

Icon'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 comprise much of what makes Icon a great language. This chapter delves into interesting features that help make Icon more than just the sum of its parts. This chapter demonstrates the following tasks:

- 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

Limiting or Negating an Expression

Chapter 1 described generators and Icon's 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.

Icon does not have a boolean type, so it might not have surprised you that 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 surrounding expression failed. The conjunction operator (`&`) resembles a short−circuit AND operator, since it evaluates its left operand, and if that operand succeeds, then 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` operator fails. The `not` operator can remedy certain forms of generator confusion. Compare the following two expressions:

```
if not (s == ("good"|"will"|"hunting")) then write("nope")
if (s ~== ("good"|"will"|"hunting")) then write("uh huh")
```

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 (~==) 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 ~== operator. You have been warned!

The conjunction operator $expr_1$ `&` $expr_2$ has an alternate syntax, a comma−separated list of expressions in parentheses: (*expr*$_1$ `,` *expr*$_2$). Any number of expressions may be present, and the whole expression only succeeds if they all succeed. This looks similar to the syntax for a procedure call because it is similar: a procedure call mutually evaluates all the actual parameters before the procedure call can proceed. Besides putting a procedure value in front of a parenthesized argument list, you can put a string or an integer. For a string value, 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.

List Structures and Parameter Lists

Built−in functions `write()` and `put()` are examples of built−in functions that take any number of arguments. These kinds of flexible functions are 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:

```
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 Icon list. If you called the above procedure with `myfunc(1, 2, 3, 4, 5)`, then `z` would have the value `[3, 4, 5]`.

It is also useful to go the other direction, that is, to construct a list data structure 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).

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. Let us consider an example. 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()
```

The `seq()` function produces an infinite sequence of integers, by default starting at 1, so the whole expression generates the sequence `"name1"`, `"name2"`, `"name3"`, ... and so forth. You can put this expression at some point in your code; but you may need to use it from several different places.

In general, there are times when you need to separate the evaluation of an expression from its textual position in the program. The normal mechanism 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. Here is a crude effort

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

Now, consider the code generating program example again. It may need not a single name sequence, but two kinds of names: statement labels and temporary variables. In this case, you will have to write a different procedure for each such sequence you need. 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 has to include a copy of the local variables and parameters used in the expression. When a coexpression is *activated*, it produces the next value. A coexpression is activated by the `@` operator. Each activation of `labelname` will produce 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
```

User−Defined Control Structures

Control structures are those elements of a language that determine in what order, and how many times, expressions are executed. Coexpressions can be used to implement new *control structures* in the sense that procedures that take coexpression arguments as parameters can control the order and number of times their arguments are activated.

Consider a control structure that selects values from the first expression at the positions specified by the second. This could be invoked as:

```
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, ?) and the primes (2, 3, 5, 7, 11, ?), this expression produces the numbers 1, 2, 5, 13, 89, ...

Here is the implementation of seqsel():

```
procedure seqsel(a)
   # We need two arguments
   (*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
```

Icon provides a syntactic short−cut for this kind of usage:

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

can also be written with curly brackets, as

```
proc{e1, e2, ..., en}
```

Parallel evaluation

Coexpressions can be used to evaluate expressions "in parallel" or in lock−step. This program writes a table of ASCII characters with the hex, decimal, and octal equivalents:

```
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
```

Now `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 coexpressions being activated once, so they are all run in lock−step, producing this table:

```
0       000     00      "\x00"
1       001     01      "\x01"
2       002     02      "\x02"
...
45      055     2d      "−"
46      056     2e      "."
47      057     2f      "/"
48      060     30      "0"
49      061     31      "1"
```

```
50      062     32      "2"
...
90      132     5a      "Z"
91      133     5b      "["
92      134     5c      "\\"
93      135     5d      "]"
94      136     5e      "^"
95      137     5f      "_"
96      140     60      "`"
97      141     61      "a"
98      142     62      "b"
...
255     377     ff      "\xff"
```

Parallel evaluation can also be used to assign to a set of variables:

```
ce := create !stat(f)
every (dev | ino | mode | link | uid | gid) := @ ce
```

Note

`stat()` returns file information. It is presented in the next chapter.

Coexpression creation can be quite expensive. This is probably not a good way to assign a group of variables to a series of values but it is a fine example to demonstrate usage.

Coroutines

In a conventional procedure invocation scenario, the procedures have an asymmetric relationship; every time control is transferred from the calling procedure to the callee, the slave procedure starts execution at the top. Coroutines have an equal relationship: when control is transferred from one coroutine to another, it starts executing from the previous point that its execution was suspended from. 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.

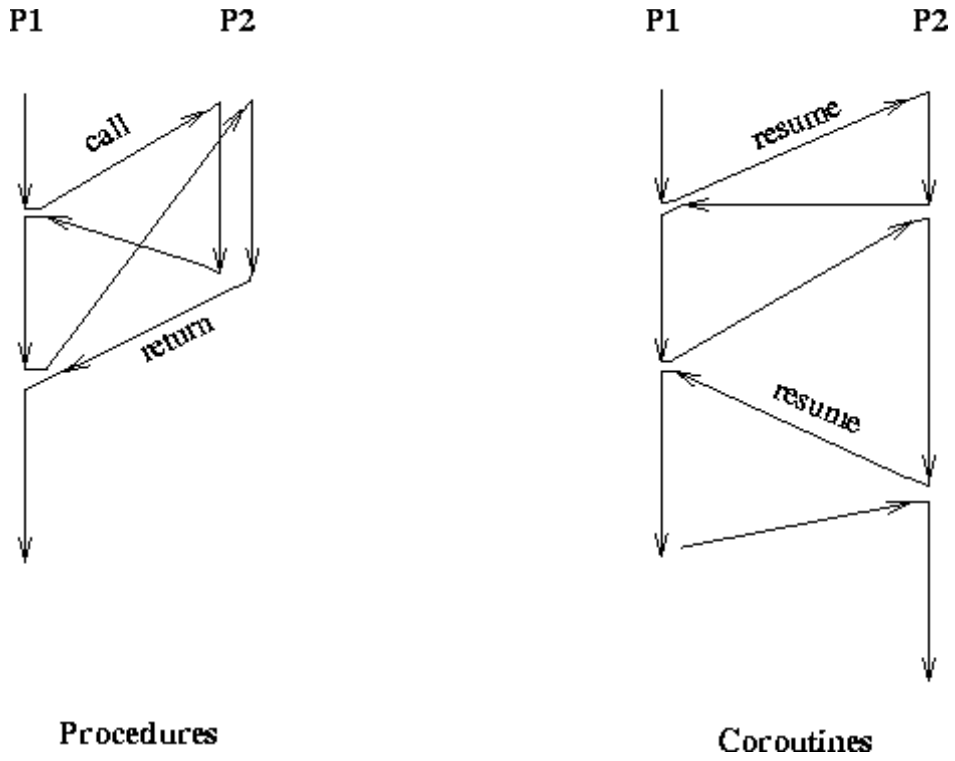

Figure 4−1:
The Difference Between Procedures and Coroutines

Here's an example: Can you tell what it computes from its integer command−line argument?

Listing 4−1
Producer and Consumer Coroutines

```
procedure main(args)
   C1 := create consumer(args[1])
   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 it a value of value; the consumer gets this value, and passes a return code (`retval`) back. The `&source` keyword is the coexpression that activated the current coexpression.

Note

The producer/consumer example should not be taken to mean that the producer/consumer problem should be done with coroutines! It is hard to find short examples that require coroutines for a clear solution.

Permutations

We have seen one usage of `map()`, where we used it to transform 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 itself is that `"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 the string `"01234"`, we therefore get the string `"edcab"` − a permutation of the string `s`! And it is exactly the permutation that is suggested by the first two arguments of `map()`. To arrange this sort of permutation, we need all three strings to be the same size, and there should be no repeated letters in the second string.

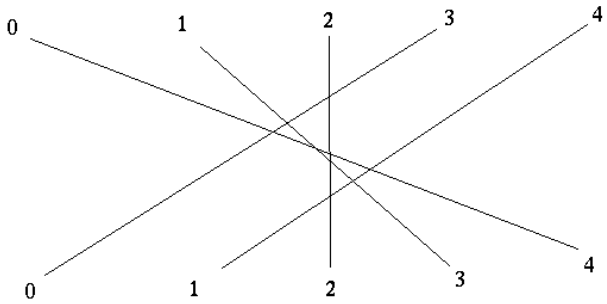

Figure 4−2:
Permuting a string with the map() function

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:

```
map("Mm/Dd/XxYy", "Dd/Mm/XxYy", date)
```

Here is another example. Icon has a built−in random facility, the `?` 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.

However, the sequence of random numbers generated by ? is the same from one run of the program run to the next. This is very good to have while we are debugging the program, because we would like it to always behave the same way while we figure what we did wrong! However, once we have it running properly, we'd like to have different runs of the program to create different numbers. Icon allows us to do this with the keyword &random − all we have to do is assign a different number to it each time, and we will get a different sequence of numbers. Here's how to assign it a number based on the current date and time:

```
&random := map("sSmMhH", "Hh:Mm:Ss", &clock) +
           map("YyXxMmDd", "YyXx/Mm/Dd", &date)
```

The effect of the above calls to map() is to remove characters from a fixed−format string quickly and easily. Now every time the program is run, the random number facility will be initialized with a different number.

Simulation

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

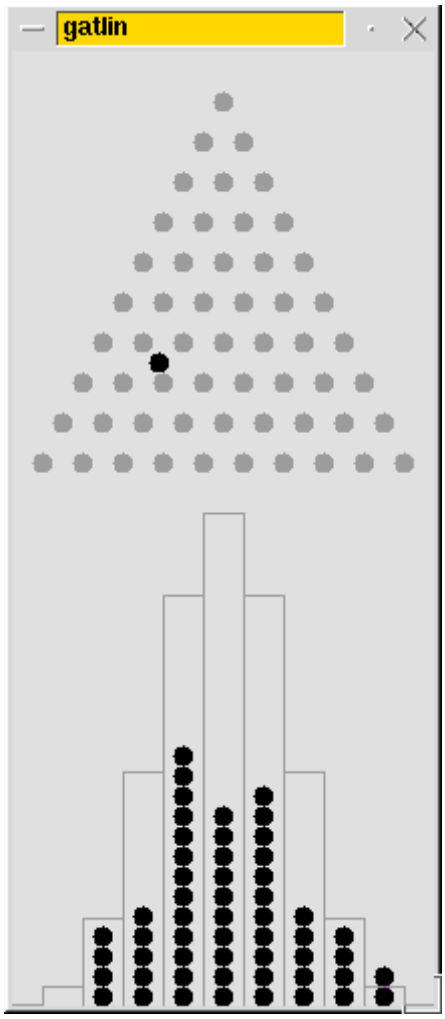

Figure 4−3:
A Galton Box Simulation

The simulation's output window is a good example of Icon's built−in high−level graphics facilities. Graphics is a broad area beyond the scope of this book; some of the examples such as this one will give you a taste of it, but you will need to consult the on−line references or the Icon graphics book (Griswold, Jeffery, Townsend 1998) for details. Graphics are part of the system interface; a brief section in the next chapter gives an overview and pointers to additional information. Some of the graphics functions used in this example include:

* `FillArc(x,y,width,height)`, which fills an ellipse (or a part of one) 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. Used with additional arguments, `FillArc()` fills portions of an ellipse similar to pieces of a pie in shape.

* `WAttrib("attr")` or `WAttrib("attr=value")`, the generic routine for getting or setting a window's 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 initial characteristics as specified by a string of 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 of the things of the screen 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 CD−ROM.)

Listing 4−2
A Simple Galton Box Simulation

```
link graphics
global pegsize, height, width, pegsize2

procedure main(args)
local n, steps
   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")
   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, y, xstep, ystep, i, lastx, lasty
   x := xfrom
   y := yfrom
   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

Icon is particularly powerful when different 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 Icon’s connection to the outside world. It refers to all input/output interactions with the host operating system. In this chapter you will learn 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

The Role of the System Interface

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 least−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. First in the graphics facilities and now in Unicon’s system interface, we have "raised the bar" of what portable facilities programmers can expect from their system interface.

The functionality described in this chapter relies on underlying standards including ANSI C’s standard input/output library, and the POSIX operating system standard. Unicon’s system interface relies on these standards, but is simpler and higher level. It is also less platform−specific than the POSIX standard. Our goal was to define facilities that can be implemented to a great extent on all modern operating systems. Icon is fairly easily ported to all systems for which ANSI C is available. Non−POSIX Unicon implementations may only provide a subset of the functionality described in this chapter, but important facilities such as TCP/IP internet communications have emerged as de facto standards and warrant inclusion in the language definition. So far we have implemented the complete Unicon system interface for Linux, Solaris, and Windows; the challenge to port these facilities to all platforms on which they will be relevant and useful now rests in the hands of Unicon’s user community.

Files and Directories

The file data type is used for any connection between a program and an external piece of hardware. This model is directly supported by most modern operating systems. In reality, a file is a reference to one or more resources allocated by the operating system for the purpose of input or output. Different kinds of connections support different operations, although most kinds of files support the basic functions given in this section.

Files are most commonly used to manipulate named repositories of data on a storage device. The contents of files exist independently from the program that creates them, and persist after that program completes its execution. 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 := open("myfile.txt", "w")
write(f, "This is my text file.\n")
```

The `open()` function takes two string parameters: a file name and a mode. The default mode is `"r"` for reading; in the preceding example the file was opened with mode `"w"` for writing. There are several other modes that denote other kinds of system interfaces. They are described in later sections.

The `read()` function automatically reads an entire line and removes the line termination character(s) before returning it. The `write()` function similarly adds a line terminator(s) after the string or strings that it writes. Another way to read lines is to use the generate operator, unary `!`. The expression `!f` generates all the lines of file `f`, so `every put(L, !f)` puts all the lines of `f` into a list named `L`.

On systems whose line terminators are not a single newline character, you can append an extra letter to the mode parameter of `open()` to indicate whether newlines are to be translated (mode `"t"`) or untranslated (mode `"u"`). Text files will generally need to be translated, while binary files will not. The default is to translate newlines to and from the 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 named collection of 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 `s`. The `open(dirname)` function will open a directory to read its contents. Directories can only be opened for reading in this way. Every `read()` from a directory returns the name of one file. Directory entries are not guaranteed to be in any specific order. The expression
`every write(!open("."))` function will write the names of the files in the current directory, one per line.

The `mkdir(s)` function creates a directory, and `rmdir(s)` deletes a directory. Only empty directories may be removed. Individual files are removed with `remove(s)`, and renamed with `rename(s1, s2)`. To remove an entire directory including its contents:

```
procedure deldir(s)
   f := open(s)
   every remove( s || "/" || ("." ~== (".." ~== !f)))
   close(f)
   rmdir(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 consists of things like the owner of the file, its size, the access rights that are granted to users, and so forth. This information is obtained with the `stat()` system call. Its argument can be either the name of a file or (on UNIX systems only) an open file. The `stat()` system call returns a record with all the information about the file. Here is a subset of ""ls", a UNIX program that reads a directory and prints out information about the files in it: Keyword `&errortext` contains the error text associated with the most recent error that resulted in an expression failure; it is written out if the attempt to open 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?

```
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's size in bytes, and `mtime`, the file's last modified time, an integer that is converted into a human readable string format by `ctime(i)`. Another important field is `mode`, a string that contains codes for the file's type and access permissions. The `mode[1]` string will be `"-"` for normal files, `"d"` for directories, and some file systems support additional types. The other characters of the mode string are platform dependent. For example 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 value of `mode[1]` 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 C includes additional 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 illustrated how different platforms' file systems vary widely in their support for the concepts of file ownership and access control. If the system supports file ownership, the user and group that own a file are changed by calling `chown(fname, user, group)`. The `chown()` function will only succeed for certain users, such as the super user. User and group may be string names or integer user identity codes; this is platform dependent.

File access rights can be changed with `chmod(fname, mode)`. The `chmod()` function will only succeed for users who own a given file. The `mode` parameter may be a nine−letter string similar to `stat()`'s mode field, or an octal integer encoding of the same information (see Appendix C for details).

Another piece of information about the files on some platforms is called the *umask*. This is a variable that tells the system what access rights any newly created files or directories should have. The function call `umask("rwxr−xr−x")` tells the system that newly created directories should have a permission of `"rwxr−xr−x"` and files should have permissions of `"rw−r?r?"`. Ordinary files are never given execute permission by the system, it must be set explicitly with `chmod()`.

File Locks

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

```
flock(filename, "x")
```

The first call to `flock()` will create a lock, and subsequent calls by other programs will cause them to block, waiting till the writing program releases its lock. The flag `"x"` represents an *exclusive* lock, which should be used while writing; this means no other process can be granted a lock. For reading, `"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.

Programs and Process Control

Unicon's system interface is similar but simpler and higher level than the POSIX C interface. An include file `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 part of the preprocessor, described in Appendix A. To include `posix.icn` in a program, add the line:

```
$include "posix.icn"
```

at the top of your program.

If a system call fails for any reason, the integer keyword `&errno` indicates the error that occurred. As seen in an earlier example, a human−readable string is also available in keyword `&errotext`..Error codes (such as EPERM, or EPIPE) are defined in `posix.icn`; `&errno` can be directly 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 that can be executed on the computer; a *process* is a program in execution. This distinction is not usually important, but when working with network applications it's important to understand the difference, since there may be more than one process running the same program, and a process can change the program that it's 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 may be trapped or ignored with the `trap(s, p)` function. Argument `s` is the string name of the signal; the list of signal names varies by platform and is given in Appendix C. You can trap any signal on any machine; if it is not defined it will be ignored. As an example, Linux systems don't have a SIGLOST, so trapping that signal has no effect when a program runs on Linux.

The `trap()` function's second argument is the procedure that should be called when the signal is received. The previous signal handler is returned from `trap()` so it can be restored using 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 operate by executing several other programs and combining their results. In many cases, you can accomplish what you need by calling `open()` with mode `"p"` (for pipe) to launch a command. In this case the string argument to `open()` is not a filename but is an entire command string. Piped commands opened for reading (mode `"p"` or `"pr"`) allow your program to read the command's standard output, while piped commands open for writing (mode `"pw"`) allow your program to write the command's standard output.

A more general function for launching programs is `system(s, f1, f2, f3)`, which runs an external command on a set of arguments, all given in the command string `s`. String `s` is parsed into arguments separated by spaces. Arguments with spaces in them may be escaped using double quotes.

The program that calls `system()` normally waits for the launched program to complete execution before continuing, and `system()` returns the integer status of the completed command. If `s` ends in an ampersand character (`&`), `system()` does not wait for the command to complete, but instead 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 illustrated by 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 identity of the process (an integer) 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 C.

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 value that gives the maximum number of milliseconds to wait before input is available. If the timeout expires, an empty list is returned. If no timeout is given, the program will wait indefinitely for input on one of the files.

```
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`:

```
   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. 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`. Using `reads()` with `select()` is a natural choice.

The command shell log application in Chapter 13 shows the usage of `select()`. Another primary application area for `select()` is network programming.

Networking

Unicon provides a very high−level interface to Internet communications. Here we will discuss both the major protocols, TCP and UDP.

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.

A call to `open()` with mode `"n"` (network) establishes 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:

```
procedure main(argv)
   local fserv

   fserv := getserv("finger") |
      stop("Couldn't get service: ", &errortext)
   name := argv[1]
   host := ""
   argv[1] ? {
      name := tab(upto('@')) & ="@" & 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 line := read(f) do
      write(line)
```

```
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:

```
procedure main()
   while f := open(":1888", "na") do {
      system("myserverd −servicerequest &", f)
      close(f)
      }
   (&errno = 0) | stop("Open failed: ", &errortext)
end
```

The call to `open(":1888", "na")` blocks until a client accesses that port. This server responds to requests by launching a separate process to handle each request. The network connection is passed to `myserverd` as its standard input, so that process had better be expecting a socket on its standard input, and must handle it appropriately. Launching a separate process to handle requests is standard operating procedure for most Internet servers, but it uses a lot of memory and CPU time. A server for simple requests that can be performed without blocking might be implemented in a single process.

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 (that is, 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 in two ways: with an `open()`/`writes()` pair, or with `send()`. Typically a server will be sending/receiving on the same socket so it will use `open()` with `read()` and `write()`. A client that is only sending one or two datagrams might use `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 `open()` signifies that a UDP connection should be used. The function `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.

```
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, you use mode `"nu"`, illustrated in the following example. 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
```

Graphics

Icon features robust high−level graphics facilities that are portable across platforms. The most robust implementations are X Window and Microsoft Windows; Presentation Manager, Macintosh, and Amiga ports are in various stages of progress. The most important characteristics of the graphics facilities are:

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

As a short example, the following program opens a window and allows the user to type text and draw freehand on it using the left mouse button, until an Escape character is pressed. Clicking the right button moves the text cursor to a new location. Mode `"g"` in the call to open stands for "graphics." The keyword `&window` serves as a default window for graphics functions. The keywords `&lpress`, `&ldrag`, and `&rpress` are special constants that denote left mouse button press and drag, and right mouse button press, respectively. The `&x` and `&y` keywords hold the mouse position associated with the most recent input event returned by `Event()`. The `"\e"` is a string containing the escape character.

```
procedure main()
   &window := open("draw example","g")
   repeat case e := Event() of {
      &lpress | &ldrag : DrawPoint(&x,&y)
      &rpress : GotoXY(&x,&y)
      "\e"    : break
      default : if type(e)=="string" then writes(&window, e)
      }
end
```

We do not cover Icon’s graphics facilities in this book; the definitive reference and programmer’s guide is "Graphics Programming in Icon" by Griswold, Jeffery, and Townsend. An electronic reference manual for the graphics facilities comes with the Icon distributions. It may also be obtained from the Icon Web site at:

```
http://www.cs.arizona.edu/icon/docs/ipd281.html
```

Summary

Unicon’s system facilities provide a high−level interface to the most common features of modern operating systems, such as directories and network connections. This interface is vital to most applications, and it also presents the main portability challenges, since Unicon has a design goal that most applications should 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 is one major area of the system interface that is a whole application area extensive enough to warrant an entire chapter in its own right: databases. 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 next chapter.

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. There are different kinds of databases, and the appropriate database to use depends on how much information is to be stored, and what kinds of accesses to the information are supported. This chapter describes three database mechanisms for which Unicon provides direct support. In this chapter you will learn how 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.

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 Icon’s 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, the term is used as the short form of the term "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 that are used to access a database determine whether a single element at a time is read or written, or whether many operations are buffered and sent to the database together.

Memory−based Databases

Some people would not consider a memory−based database to be a database at all. If the entire database fits in memory at once, you can generally achieve vast speedups by avoiding the disk as much as possible. For example, all forms of querying or reading 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 were limited in

value ten years ago, but in the age where a gigabyte of main memory for your PC is affordable, they may be the best choice for many or most applications.

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

```
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:

```
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 on−going basis and for which the queries can readily be expressed as operations on Icon's 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.

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 Icon's table data type. DBM was eventually followed by compatible superset libraries called NDBM (New Data Base Manager) and GDBM (GNU Data Base Manager). The availability of a GNU source code implementation almost ensures that DBM can be supported on any platform.

DBM databases are opened by the `open()` function using mode `"d"`. By default, a database is opened for reading and writing; mode `"dr"` opens a database read−only. Once opened, DBM databases are treated as a special case of 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 just dereferenced for its current value. For persons who do not like to use subscript operators for everything, 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.

```
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:

```
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. In fact, Netscape uses DBM files for its bookmarks.

There is one basic limitation of DBM databases when compared with Icon's 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, an Icon 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 `xencode()` and `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.

SQL Databases

DBM is great for data that can be organized around a single key, but it is not much help for more complex databases. The industry choice for enterprise−level data organization revolves around the Structured Query Language (SQL). SQL is supported by every major database vendor.

A SQL database is more complex than a DBM database. It 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.

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 *tables*, and each table is a collection of *rows*. The rows in a table contain information of various types in a set of named *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, but is now primarily used "under the covers" by database applications. Many database applications 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 CREATE TABLE statement, such as

```
create table addresses (name    varchar(40),
                        address varchar(40),
                        phone   varchar(15))
```

Tables have a primary key, which must be unique among rows in the table. By default the primary key is the first one listed, so `name` is the primary key in table `addresses` defined above.

SQL's data manipulation operations include SELECT, INSERT, UPDATE, and DELETE. SELECT allows you to refine the data set being operated on, picking rows and columns that form some projection of the original table. Perhaps more importantly, SELECT 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:

```
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

```
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.

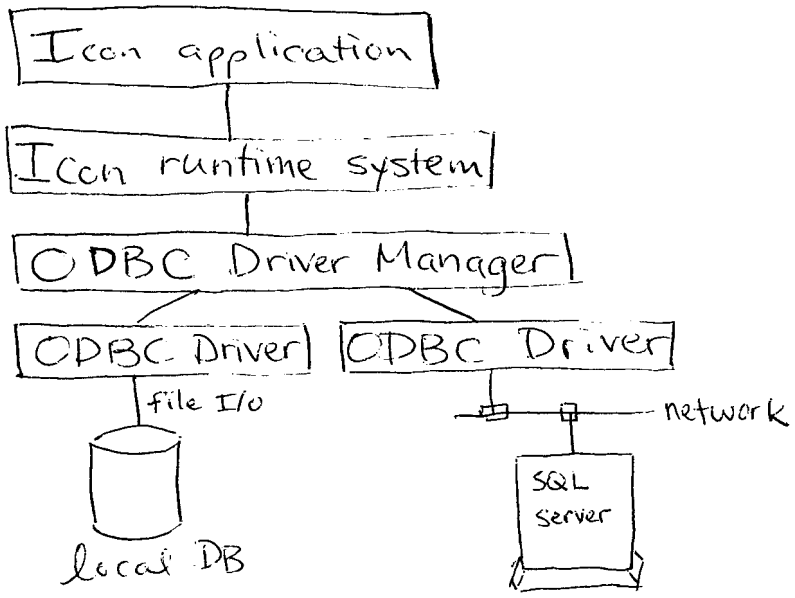

Figure 6−1:
ODBC hides the database architecture, and Icon hides ODBC from applications

To use Unicon's SQL facilities, you must have several software components in place. First, you need A SQL server that supports ODBC. You could buy a commercial SQL server, or you may be able to use a free or low−cost server such as MySQL (`www.mysql.org`) or PostgreSQL (`www.postgresql.org`).

Second, you need an account, password, and permissions on the SQL server, so that you can connect to it. The details of this process are necessarily server−dependent and unfortunately outside the scope of this book.

Third, your client machine needs an ODBC driver manager, and an ODBC driver for your SQL server, and these have to be configured properly. The driver manager is a

piece of system software that multiplexes applications to different databases, while the drivers are the dynamic link libraries that database vendors supply to talk to their database. Once you have the ODBC software set up, writing the Unicon client program to connect to your database is straightforward.

Opening A SQL Database

Connecting to a SQL database is performed by calling `open()` with mode `"q"`. 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 "q" uses additional arguments to `open()` that specify the table, 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:

```
f := open("unicondb", "q", "mytable", "scott", "tiger")
```

The `open()` function returns a value that supports many of the operations of both a file and a table. If the connection can't be established, the call fails. The underlying session information is maintained separately and may be shared by multiple calls to `open()` to 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 a SQL Database

Subsequent queries to the database can be made by calling `sql(db, sqlstring)`. 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(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

When you wish 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

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

Called with one argument, `fetch(db)` moves the cursor forward one position. With two arguments, fetch(db, column) fetches a single column from the current row, without advancing the cursor. The function `seek(db, x)` moves the cursor relative to the current query selection, without fetching any data. If `x` is an integer or convertible to one, the `seek()` function is positional within the set of selected rows. The first position in a set of rows is position 1, like the other Icon sequence types. If `x` is not an integer, it must be a string, and the resulting seek is associative. The cursor moves forward until a column value matches `x`. If the string begins with a column name and an equals sign, the seek matches that column against the remainder of `x` after the equals sign. If `x` does not begin with a column name and equals sign, the seek matches `x` against the primary key.

Dynamic Records

You may be wondering exactly how rows are represented. Actually, they are records, but their fields are determined at run−time from the names of selected columns. Such new record types that are introduced on the fly are called *dynamic records*, and they are useful in other contexts besides databases.

The function `constructor(recordname, field, field, ?)` produces a Unicon record constructor procedure for records named `recordname` with the specified fields. The fields of dynamic records can be arbitrary strings, not just legal identifiers, but obviously the field operator (`.`) cannot be used for fields that are not identifiers.

Updating a SQL Database

Updates to the database are made using the `update()` function. The following example updates two columns within a row returned by `fetch()`.

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

Inserting and deleting rows are performed by functions `insert()` and `delete()`. The `insert()` function takes two parameters for each column being inserted, the column name and then the value.

It is worth mentioning that many forms of SQL selections are read−only. The relational combination of columns from different tables is a powerful abstraction, 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 guaranteed to be updatable, but if you do something fancier, you will have to know the semantics of SQL to tell whether the result may be modified.

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’sID,

the pay rate, a code indicating whether pay is hourly or salaried, and the job title. Note that the SQL is embedded within an Icon string literal. An Icon string spans multiple lines when the closing doublequote has not been found and the line ends with an underscore character.

```
sql("create table employees_
   (id varchar(11), name varchar(40), phone varchar(15))")
sql("create table jobs_
   (id varchar(11), payrate integer, is_salaried char,_
    title varchar(40))")
```

Inserting rows into the database is as simple as

```
insert(db, "id", 32, "name", "Ray", "phone", "274-2977")
```

Note the alternating column names and values. This call to `insert()` is equivalent to the SQL statement, which could be transmitted by a call to `sql()`:

```
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:

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

SQL Types and Icon Types

SQL has many data types, most of which correspond closely to Icon types. For example, 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.

Icon 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 Icon. These types, such as bit strings, datetimes, timestamps, and BLOBS, are generally all represented by Icon strings, and Icon strings are used to insert such values into SQL databases. In fact, Icon strings are the "magic fingers" of the whole interface, since they are also the type used by the system to represent out of range values when reading SQL columns into Unicon. But, you are probably tired of me stringing you along about databases by now.

More SQL Database Functions

SQL databases are complicated enough, and feature-rich enough to warrant their own repertoire of functions in addition to the operations they share with other kinds of files

and databases. These functions are described in detail in Appendix A, but a few 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 `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 `dbproduct(db)` and `dbdriver(db)` produce information about the DBMS to which db is connected, and the ODBC driver software used in the connection, respectively. The function dblimits(db) produces the upper bounds for many DBMS system parameters, such as the maximum number of columns allowed in a table. All three of these functions return their results as either a record or list of records; the record field names and descriptions are given in Appendix A.

Summary

Databases are a vital form of persistent storage for modern applications. The notation used to manipulate a database can look like a sequence of table and record operations, or it may be an obvious combination of Unicon and SQL statements. Either way, database facilities give programmers more direct access and control over the information stored in permanent storage.

Part II: Object−oriented Software Development

Chapter 7: The Object−Oriented Way of Thinking

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 languages that are not considered object−oriented. The next several chapters present these ideas and illustrate their use in design diagrams and actual code. Both diagrams and code are relevant because they 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

Objects in Programming Languages

Object−oriented programming can be done in any language, but some languages make it much easier than others do. Support for objects should not come at the cost of strange syntax or programs that look funny in a heterogeneous desktop−computing environment. Smalltalk has these problems. C++ is based on a well−known syntax and is used for mainstream desktop computing applications, but its low−level machine−orientation makes C++ less than ideal as an algorithmic notation usable by non−experts. Java offers a simple object model and a familiar syntax − a potent combination. The main advantages Unicon has over Java are its fundamentally higher−level built−in types, operations, and control structures.

Many object−oriented languages require that *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 did not originally support user−defined objects, although the built−in structure types have nice object−like encapsulation and polymorphism properties. Unicon's object−oriented facilities descend from a package for Arizona 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.

Key Concepts

This section describes the general concepts that object−orientation introduces 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 as a guideline, 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 (or can be, or ought to be) irrelevant to that inside, and vice versa. All communication across the boundary occurs through a public interface. We call an encapsulated data structure an object. Just as a set of named variables called parameters comprises the only interface between a procedure and the code that uses it, a set of named procedures called methods comprises the only 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 new data structure that is similar to a structure that is already written, one specifies the new structure by giving the differences between it and the old structure, instead of copying and then modifying the old structure's code. Obviously 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 a variety of common 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.

In fact, we can make a stronger statement. Due to encapsulation, two different executions of some code that uses objects to implement a particular algorithm may operate on different 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.

Objects in Program Design

Another fundamental way to approach object−oriented thinking 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.

The Unified Modeling Language (UML) is a diagramming notation for building software models. It was invented by Grady Booch, Ivar Jacobson, and James Rumbaugh of Rational Software. In UML, software models document the purpose and method of a software system. For small, simple, or well−understood software projects, a prose description may be all the documentation that is needed. UML is of interest to designers of larger software systems for which a prose description alone would be inadequate. 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 would use these kinds of diagrams in sequence. You might 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 manipulated by the system. Finally, construct statechart diagrams if needed to describe the sequences of events that can occur and the corresponding operations performed by various objects in response to those events.

While use case and statechart diagrams are important, their purpose is to help organize and elaborate on the object model represented by one or more class diagrams. For this reason, the next two chapters present elements of class diagrams along with the corresponding programming concepts. A discussion of use case diagrams and statecharts is given in chapter 11.

Summary

Unicon provides object−oriented programming facilities as a collection of tools to reduce the complexity of large programs. These tools are encapsulation, inheritance, and polymorphism. Since a primary goal of Unicon is to promote code sharing and reuse, various specific programming problems have elegant solutions available in the Unicon class library.

Chapter 8: Classes and Objects

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. In this chapter you will learn how to:

- Draw diagrams that show class names, attributes, and methods
- Write corresponding code for classes and their methods
- Create instances of classes and invoke methods on those objects

Class Diagrams

Modeling a software system begins with the task of identifying things that are in the system and specifying how they are related. A class diagram shows a static view of relationships between the main kinds of elements that occur in the problem domain. A class diagram is a data−centric, object−centric model of the program. 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 8−1 shows an example class.

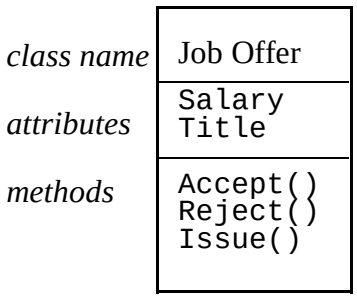

Figure 8−1:
Class Job Offer has two Attributes and Three Methods

When an object model is implemented, classes in the 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 8−2 shows two very different levels of detail for the same class. Each level of detail is appropriate in different contexts.

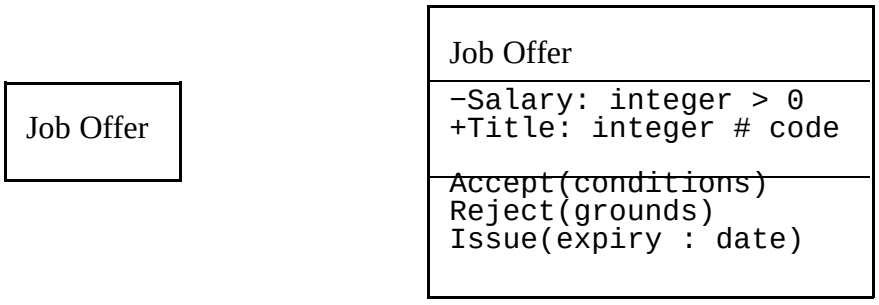

Figure 8−2:
A Class is Drawn with Different Levels of Detail in Different Diagrams

You can draw rectangles with names of classes inside them all day, but unless they say something about program organization, such diagrams serve little purpose. The main point of drawing a class diagram is not the classes; it is the relationships between classes that are required by the model. These relationships are called *associations*. In a class diagram, associations are lines that connect classes. Accompanying annotations in the diagram convey relevant model information about the relationships. The details of associations and their implementation are described in chapter 9.

Declaring Classes

In Unicon program code, the syntax of a class is:

```
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*. The syntax of a method is that of a procedure:

```
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 Icon procedure, global, and record declarations; such declarations have the standard semantics and exist in the global Icon name space.

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 their 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:

```
procedure main()
   f := foo(1, 2)
end
```

The fields of an object need not be initialized by a parameter passed in to the class constructor. For many objects it is more logical to initialize their 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 method with no parameters that is invoked automatically by the system.

An `initially` section begins with a line containing the reserved word `initially`, followed by lines of code that are executed whenever an object of that class is constructed. These lines typically assign values to 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:

```
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 class constructor:

```
procedure main()
   mytaque := taque()
end
```

In the absence of an `initially` section, missing arguments to a constructor default to the null value. Together with an `initially` section, the class declaration looks like a procedure that constructs objects of that class. Note that you may write classes with some fields that are initialized explicitly by the constructor while other fields are initialized automatically in the `initially` section. In this case you should either declare the automatically initialized fields after those that are initialized in the constructor, or insert `&null` in the positions of the automatically initialized fields in the constructor. The parameters to the constructor are 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. The `initially` section is really just a special method, and it is allowed to take a parameter list just like any other method. 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 class, the third field is initialized from constructor parameter `x`, overridding the default behavior of initializing fields in the declared order. This capability becomes important in large classes with many fields.

```
class C(a, b, c)
initially(x)
   c := x
end
```

Object Invocation

Once you have created an object with a class constructor, you manipulate the object by invoking methods defined by its class. Since objects are both procedures and data, object invocation is a combination of a procedure call and a record access. The syntax is

```
object . methodname ( arguments )
```

If an object's class is known, object methods can also be called from Icon using a normal procedure call. Called as an Icon 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.method(arguments)` is equivalent to `C_method(x, arguments)`.

Although object methods can be called using Icon 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:

```
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 properly access an object. Although an object is a kind of record, direct 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 behind the covers, but attributes and methods are referenced by name, like other variables. The taque insert method is thus:

```
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.

Comparing Records and Classes by Example

The concepts of classes and objects are found in many programming languages. The following example clarifies the object model adopted by Unicon and provides an initial impression of these concepts' utility in coding. To motivate the constructs provided by Unicon, 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 Icon structures holding the contents of various text files. You might begin with a simple structure like the following:

```
record buffer(filename, text, index)
```

where `filename` is a string, `text` is a list of strings corresponding to lines in the file, and `index` is a marker for 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 no doubt write Icon 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 kept in a separate source file (`buffer.icn`) and understood independently of the program(s) in which they are used. Here is one such procedure:

```
# 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. This example lays the groundwork for some more substantial techniques to follow. Methods are like procedures that are always called in reference to a particular object (a class instance).

```
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 makes for shorter names. The same method name, such as `read()` in the above example, can be used in each class that implements a given operation. This notation is more concise than is possible with standard Icon procedures. More importantly it allows an algorithm to operate correctly upon 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 observations in 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 also illustrates an important scoping issue. Within class buffer, method `read()` effectively makes the regular Icon built−in function `read()` inaccessible! You should be careful of such conflicts. It would be easy enough 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 Icon 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.

Summary

Classes are global declarations that encompass a record data type and the set of procedures that operate on that type. Instances of a class type are called objects, and are normally manipulated solely by calling their associated methods. Within a class, an object’s fields are accessed as variables without the usual record dot notation, and an object’s methods may be invoked by name, with an appearance similar to ordinary procedure calls.

Chapter 9: 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 from Chapter 2 to implement associations

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 `fraction(numerator, denominator)`, you could define class `inverses(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 9−1, in which an offer of a salaried appointment is defined as one kind of job offer. The attributes (salary, title) and methods (Accept() and Reject()) of class JobOffer are inherited by class SalariedAppointment, and do not need to be repeated there. A new attribute (term) is added in SalariedAppointment that is not in JobOffer.

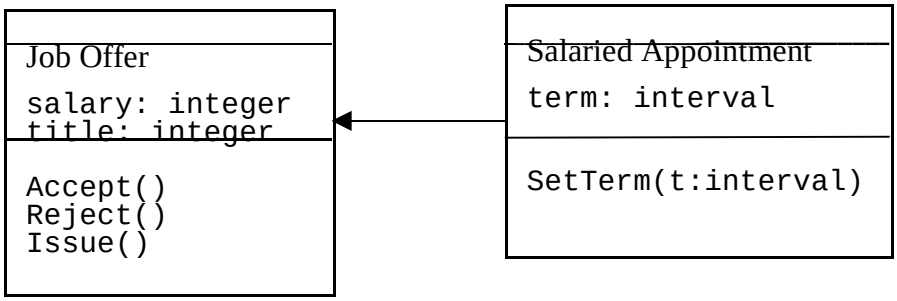

Figure 9−1:
A Salaried Appointment is a subclass of a Job Offer

The syntax of a subclass is

```
class classname superclasses (attributes)
   methods
initially_section
end
```

Where *superclasses* is an optional list of class names separated by colons, *attributes* is an optional list of variable names separated by commas, *methods* is an optional list of declarations for the class methods, and the *initially_section* is optional initialiation 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.

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 all 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 addition of fields and methods follows a simple linear bottom−up examination 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. In Unicon, multiple inheritance adds fields and methods in an order defined by a depth−first traversal of the parent edges of the superclass graph. This is discussed in more detail later on. For now, 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 either by parameters to the constructor or by the class initially section. Initially sections are methods and are inherited in the normal way. It is common for a subclass initially section to call one or more of their superclasses' initially sections, for example:

```
class sub : A : B(x)
initially
   x := 0
   A.initially()
   B.initially()
end
```

It is common to have some attributes initialized by parameters, and others within the initially section. For example, to define a class of inverse values (numbers of the form 1 / *n* where *n* is an integer) in terms of a class fraction(numerator, denominator) one would write:

```
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 always result in 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 examples of inheritance can be obtained by extending the buffer example from the previous chapter. Suppose the application is more than just a text editor: it includes word−associative databases such as a dictionary, bibliography, spell−checker, and thesaurus. These various databases can be represented internally using Icon tables. The table entries for the databases vary, but the databases all use a 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 9−2 shows a class diagram with several subclasses derived from class `buffer`. A class named `buftable` refines the buffer class, adding support for random access. Three other classes are defined as subclasses of `buftable`. The implementation of these classes is given below.

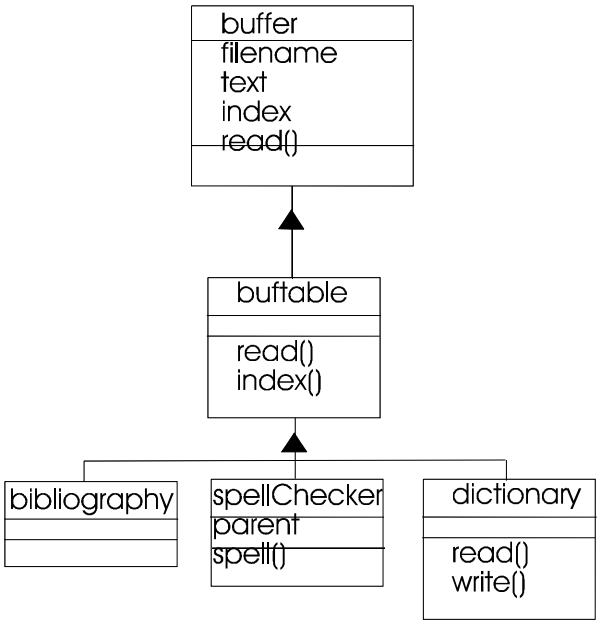

Figure 9−2:
Some Subclasses of the Buffer Class

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.

```
class buftable : buffer()
  method read()
    self.buffer.read()
    tmp := table()
    every line := !text do
      line ? { tmp[tab(many(&letters))] := line | fail }
    text := tmp
    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 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 buftable class. An example keyword would be Jeffery99.

```
class bibliography : buftable()
end
```

Spell−checkers

The database for a spell−checker might be just a list of words, one per line, the minimal structure required by the buftable class given above. Some classes exist to introduce new terminology rather than to define a new data structure. In this case we introduce 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, we allow spellChecker objects to chain together for the purpose of looking up words.

```
class spellChecker : buftable(parent)
  method spell(s)
    return \text[s] | parent.spell(s)
  end
end
```

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.

```
class dictionaryentry(word, pos, etymology, definition)
  method decode(s) # decode a dictionary entry into its
components
    s ? {
      word       := tab(upto(';'))
      move(1)
      pos        := tab(upto(';'))
      move(1)
      etymology  := tab(upto(';'))
      move(1)
      definition := tab(0)
    }
  end
```

```
  method encode()  # encode a dictionary entry into a string
    return word || ";" || pos || ";" ||
              etymology || ";" || definition
  end
initially
  if /pos 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
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).

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 degree d and radian r. If one were implementing classes corresponding to these types there is no reason why one of them should be considered a subclass of the other. The types are truly interchangeable and equivalent.

In Unicon, expressing this equivalence is simple and direct. In defining classes Cartesian and Radian we may declare them to be superclasses of each other:

```
class Cartesian : Radian (x, y)
# code that manipulates objects using Cartesian coordinates
end

class Radian : Cartesian (d, r)
# code that manipulates objects using radian coordinates
end
```

These superclass declarations make the two types equivalent names for the same type of object; after inheritance, instances of both classes will have fields x, y, d, and r, and support the same set of operations.

Equivalent types each have a unique constructor given by their class name. Often the differing order of the parameters used in equivalent types' constructors reflects different points of view. Although they export the same set of operations, the actual procedures invoked by equivalent types' instances may be different. For example, if both classes define an implementation of a method print(), the method invoked by a given instance depends on which constructor was used when the object was created.

If a class inherits any methods from one of its equivalent classes, it is responsible for initializing the state of all the fields used by those methods in its own constructor. It is also responsible for maintaining the state of the inherited fields when its methods make state changes to its own fields. In the geometric example given above, for class `Radian` to use any methods inherited from class `Cartesian`, it must at least initialize `x` and `y` explicitly in its constructor from calculations on its `d` and `r` parameters. In general, this added responsibility is minimized in those classes that treat an object's state as immutable.

Associations

An association denotes a relationship between classes that is manifested by objects at runtime. In the same sense that 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.

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 9−3 shows an aggregate found in the domain of sports, where a team is comprised of a group of players.

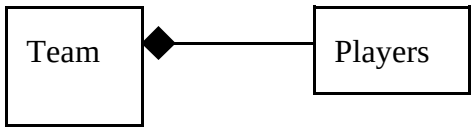

Figure 9−3:
A Team is an Aggregation of Players

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.

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 9−4 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.

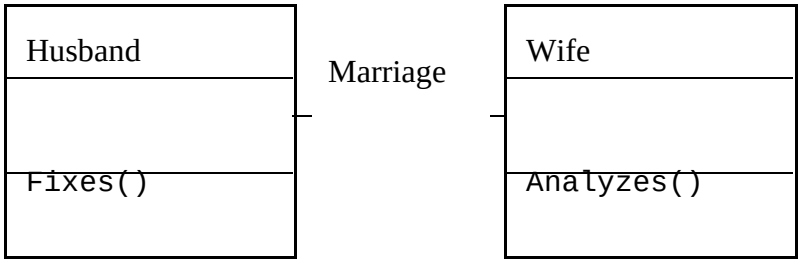

Figure 9−4:
A User−Defined Association

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 9−5 shows a BasketballTeam that is an aggregate with a multiplicity of five Players.

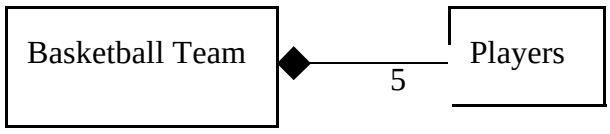

Figure 9−5:
Multiplicity of a Basketball Team

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 different 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 9−6 shows a better model of the classes and roles involved in a Marriage association depicted earlier in Figure 9−4.

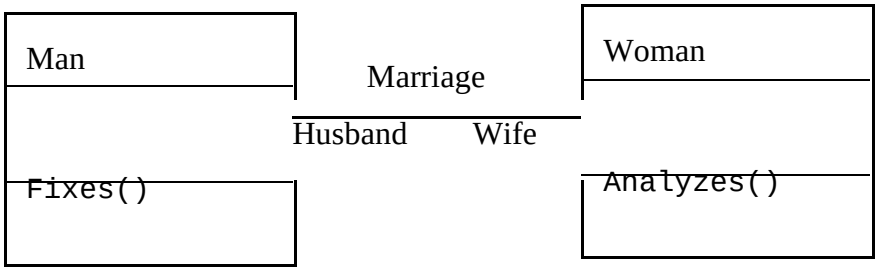

Figure 9−6:
Roles in an Association

A *qualifier* is a key value used to efficiently distinguish and directly access 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 9−7 shows a reinterpretation of the basketball aggregation in which the players on the team are distinguished using a qualifier key named position.

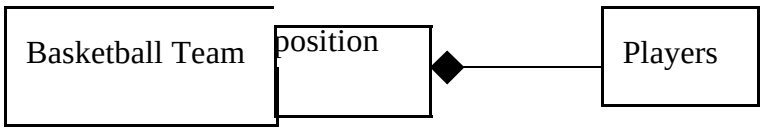

Figure 9−7:
Using a Qualifier in Lieu of Multiplicity

Implementing Associations

Unlike inheritance, which is implemented by the language and resolved at compile time, associations involved 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 Icon's built−in structure types instead of writing classes for them.

In all cases, associations will introduce additional fields into the classes being associated. The following code example implements the Marriage association from Figure9−6. 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.

```
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 may be implemented using an Icon table. The following code example corresponds to the basketball team diagram in Figure 9−7. 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.

```
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.

```
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 may be reserved for rare cases such as many−many relationships and associations that have their own state or behavior. Extended examples of associations and their implementation are given in Part 3 of this book.

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. Icon’s structure types allow most associations to map very naturally onto code.

Chapter 10: Writing Large Programs

This chapter describes language features, techniques, and software tools that play a supporting 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 make large programs smaller through code reuse.

In the case of Unicon, "large" might mean any program of sufficient complexity that it is not self−explanatory without any special effort. This includes programs over a few thousand lines in size, as well as all programs 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 or sentences 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:

- Use design patterns to simplify and improve software designs
- Organize programs and libraries into packages
- Generate HTML indices and reference documentation for your code

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 is in its infancy, since 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.

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.

```
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 10−1 shows a proxy serving a client object in lieu of the real object.

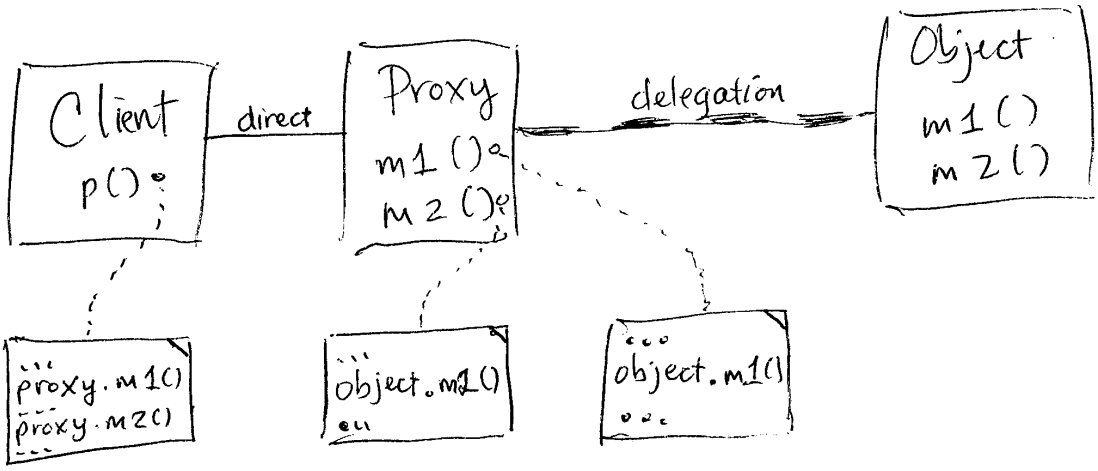

Figure 10−1:
The Proxy Pattern

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.

```
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()
   end
   method depart()
      g := &null
   end
end
```

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.
   # Don’t need an instance of slow if user only needs these
values.
   t[1] := 1
   t[2] := 1
end
```

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 10−2 illustrates the visitor pattern.

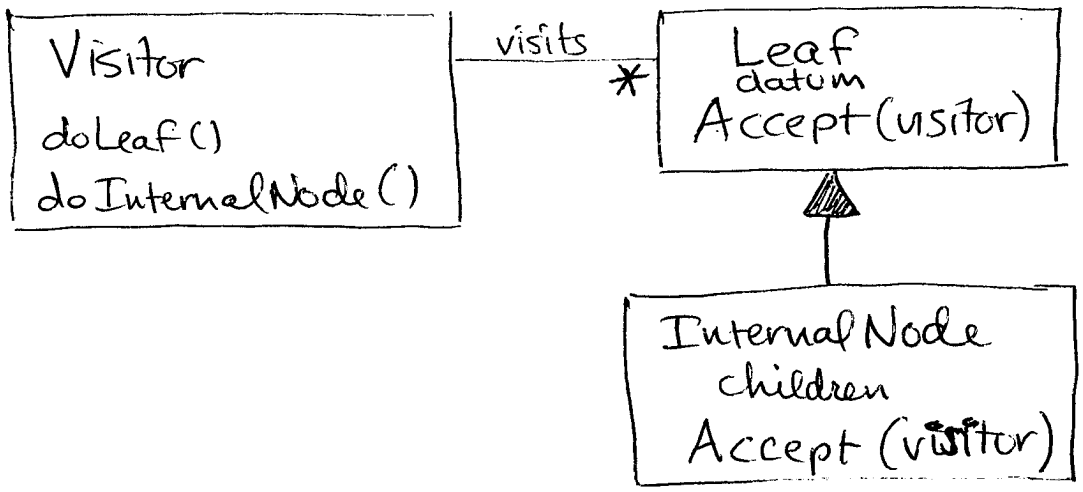

Figure 10−2:
The Visitor Pattern

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.

```
class Visitor()
   method DoLeaf(theLeaf)
      # ... visit/use theLeaf.datum
   end
   method DoInternalNode(theNode)
      # ... visit/use theNode.datum
   end
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
```

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 class with only one instance. Every global declaration (variables, procedures, records, and classes) is "invisible" outside the package.

Package membership is declared by including a package declaration in an Unicon source file; this specifies that the file belongs to that package. The package declaration looks similar to the link declaration. You can provide an identifier or a string filename:

```
package foo
```

or

```
package "/usr/local/lib/icon/foo"
```

Global names from the same package are referenced normally. Global names outside the package are not visible by default. To write out a variable `v` in package `foo`, you can say

```
write(foo.v)
```

or you can import package `foo`, making all its names visible in the current source file.

```
import foo
...
write(v)
```

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.

Name Conflicts

The purpose of packages is to reduce name conflicts, especially accidental ones. You will get a link error if you try to 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. Note, though, that it is possible to compile files out of sequence in such a way that this mechanism is defeated. Suppose you compile file `A,` which is in package `foo`; then you compile file `B,` which imports `foo` and declares variable `V`; and then you compile file `C,` which is in package `foo` and also declares `V`. Instead of an error, the `V` in file `B` will be a different variable from the `V` in file `C`, until the next time you recompile file `B`, at which point the compiler will see the imported variable `V` and will produce an error message for the redeclared variable. The moral of the story is that you should compile all files in a package before you import that package.

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.

Name Mangling

It is worth mentioning that 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`.

HTML documentation

IconDoc is an Icon documentation generator, inspired loosely by Java's JavaDoc program, and based on an HTML−generating program called iplref, by Justin Kolb. IconDoc depends on your program being in "IPL normal form", which is to say that comments in your source files should in the format used in the Icon Program Library. From these comments and the signatures of procedures, methods, records, and classes, IconDoc generates reference documentation in HTML format.

The advantage of this approach is that it produces reference documentation automatically, without altering the original source files. Run IconDoc 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.

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 11: Use Cases, Statecharts, and Collaboration 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 8 and 9 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 the states that govern an object's behavior, 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.

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 11−1.

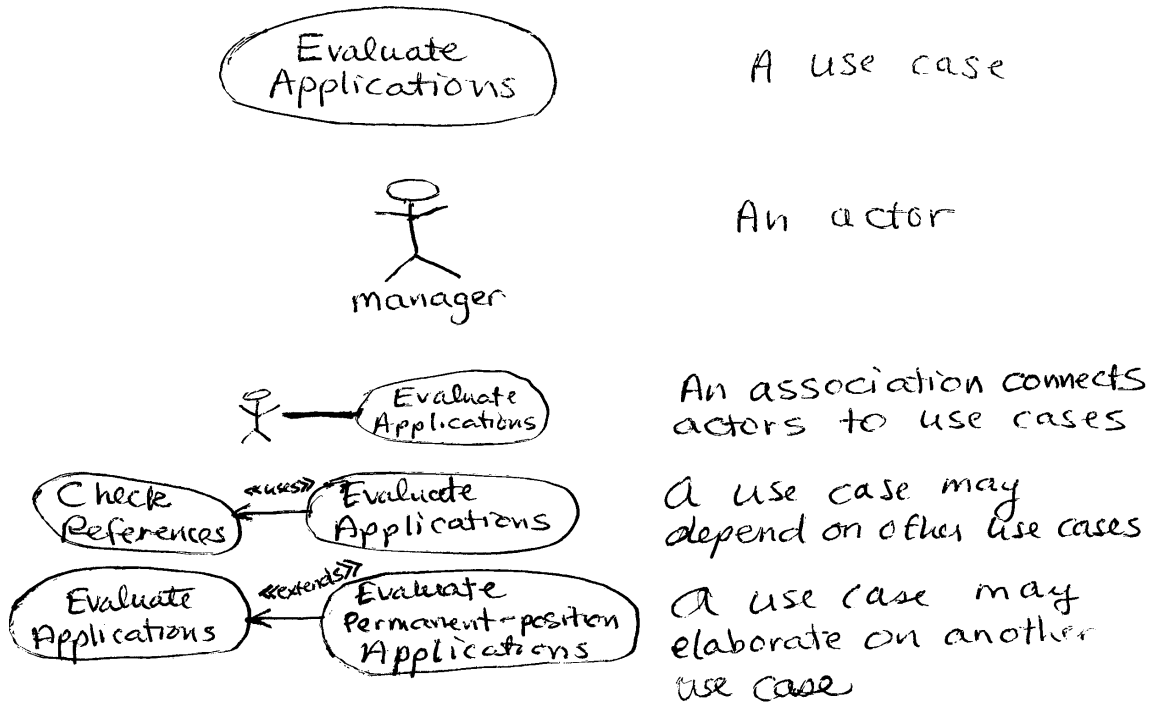

Figure 11−1:
The main components of use case diagrams

The *use cases* themselves are shown as ovals. The name of the use case goes inside the oval. The use cases have an accompanying description. An example use case 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 entirely by one actor, or may involve several actors. Use case associations identify the actors that participate in each use case. They are only distantly related to the associations between classes found in class diagrams.

Dependencies and *elaborations* between use cases are drawn as lines with arrows, and annotated with a label between the << and >>. Some use cases use (rely on) other use cases as part of a more complex task. Other use cases are best defined as extensions of an existing 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 11−2, 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.

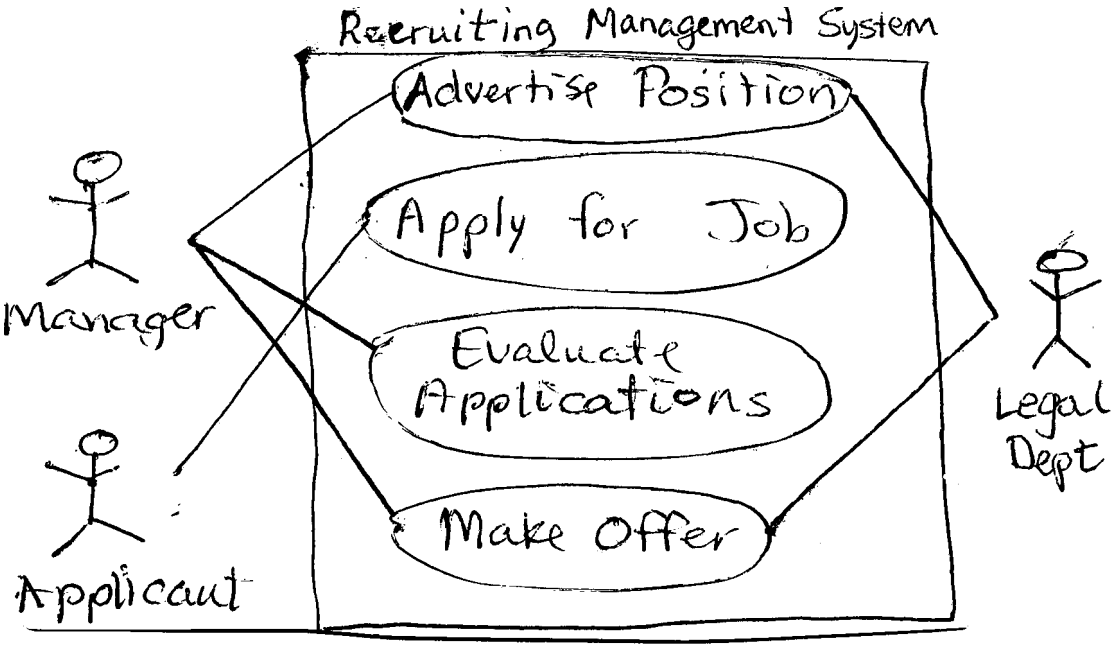

Figure 11−2
A Use Case Diagram

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 may be more detailed than this example. For example, one field might consist of the most common sequence of events, emphasized by an explicit enumeration. The most common variations on the primary

event sequence are also of value. A more organized description of the Make Offer use case might look like

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.

Statechart Diagrams

Statecharts are diagrams that represent 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 one state to another. 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 11−3 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.

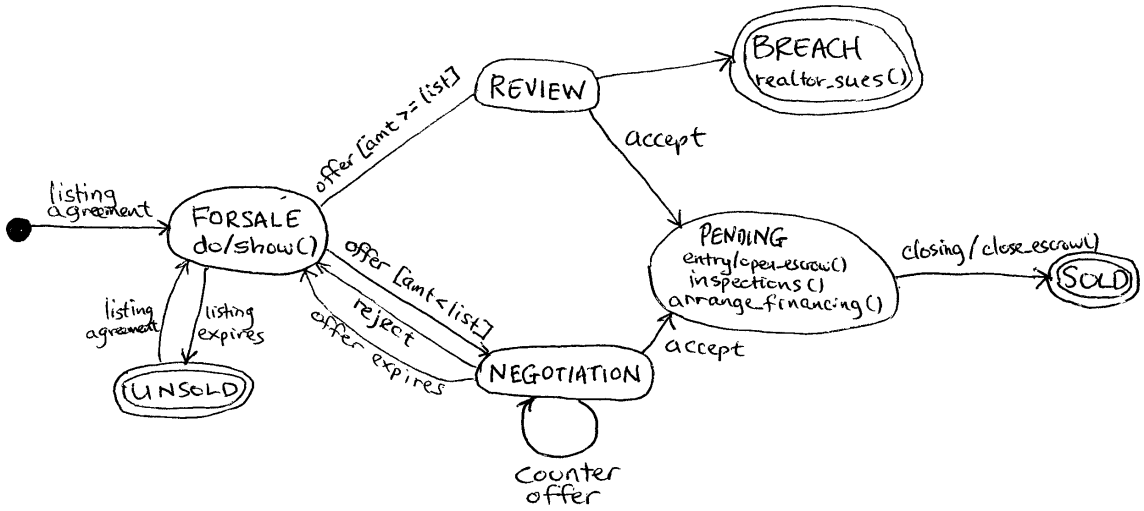

Figure 11−3:
A Statechart Diagram

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 11−3 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 identify a transition that can only occur if a particular Boolean condition is satisfied. In Figure 11−3, the event offer was used for two transitions out of the same state, with different conditions (amount >= 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]`.

If you have 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 fine 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 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 character and an action name after the event name (and after the condition, if there is one). 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 continuous or repeated activity. In Figure 11−3, the activity `do / show()` indicates 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.

Collaboration Diagrams

Statecharts are normally used to model the state of an individual 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 with many objects, it is useful to understand the interactions among 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 11−4 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.

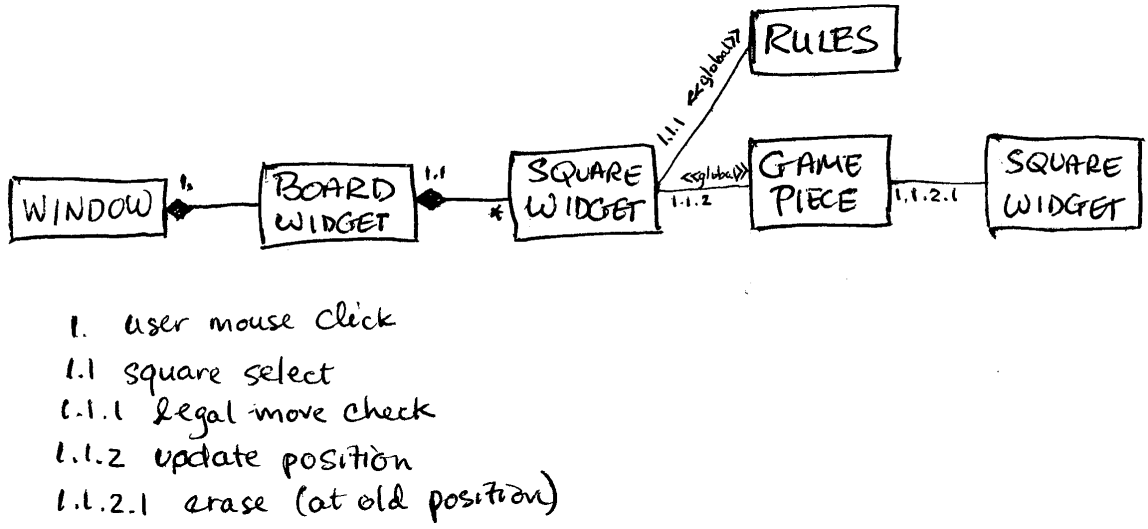

Figure 11−4:
A Collaboration Diagram

There are a couple more annotations worth noting in Figure 11−4. 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 <<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 <<global>> you can annotate a link as a <<parameter>> or <<local>> to indicate other non−association references through which messages are transmitted.

Summary

This chapter introduced four diagramming techniques from the UML that are useful in modeling dynamic aspects of a program's behavior. To learn more about these techniques and several others, you may want to study a book with a primary emphasis on UML, such as *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.

Part III: Example Applications

Chapter 12: CGI Scripts

CGI scripts are programs that read and write information to process input forms and generate dynamic content for the World Wide Web. CGI programs are called "scripts" because they are often written in scripting languages, but they can be written in any language, such as C. The Icon programming language 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

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.

This chapter describes `cgi.icn`, a library of Icon 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 Icon 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 `cgi.icn`, place the statement `link cgi` at the top of your Icon 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 `<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 `</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 `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, `cgi.icn` includes a `main()` procedure that processes the input and writes HTML header and tail information around your program's output.

> **Note**
>
> When you use `cgi.icn`, you must call your main procedure `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 `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:

```
<INPUT TYPE = "text" NAME = "PHONE" SIZE=15>
```

which allows input of a string of length up to 15 characters into a field named `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 an Icon global variable named `cgi`. The keys of this table are exactly the names given in the HTML `INPUT` tags. The values accessed from the keys are the string values supplied by the user. For example, to access the `PHONE` field from the above example, the application could write

```
cgi["PHONE"]
```

Processing Output

The main task of the CGI program is to write an HTML page to its standard output, and for this task `cgi.icn` provides a host of procedures. Typically these procedures convert an Icon value into a string, wrapped with an appropriate HTML tag to format it properly. A typical example is the library procedure `cgiSelect(name,values)`, which writes an HTML `SELECT` tag for a field named name. The `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 `cgiSelect()`. An Icon programmer might write

```
cgiSelect("GENDER", ["female", "male"])
```

to generate the HTML

```
<SELECT NAME="GENDER">
<OPTION SELECTED>female
<OPTION>male
</SELECT>
```

Common CGI Environment Variables

Much of the official CGI definition consists of a description of a set of standard environment variables that are set by the Web server as a method of passing information

to the CGI script. Icon programmers access these environment variables using `getenv()`, as in

```
getenv("REMOTE_HOST")
```

Table 12−1 contains a summary of the CGI environment variables is provided 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 http://hoohoo.ncsa.uiuc.edu/cgi/env.html on the Internet.

Table 12−1
CGI Environment Variables

Variable	Explanation
`CONTENT_LENGTH`	The length of the ASCII string provided by `method="POST"`.
`HTTP_USER_AGENT`	The user's browser software and proxy gateway, if any. The format is *`name/version`*, but varies wildly.
`QUERY_STRING`	All of the information which follows the ? in the URL when using `method="GET"`. This is the string that holds all of the information submitted through the form. Since `QUERY_STRING` is parsed and made available through a table stored in the global variable `cgi`, programmers that use `cgi.icn` do not generally consult this environment variable.
`REMOTE_ADDR`	The IP address of the client machine.
`REMOTE_HOST`	The hostname of the client machine. Defaults to IP held by `REMOTE_ADDR`.
`REQUEST_METHOD`	The method, either `GET` or `POST`, that was used to invoke the CGI script.
`SERVER_NAME`	The server's hostname. It defaults to the IP address.
`SERVER_SOFTWARE`	The Web server that invoked the CGI script. The format is *`name/version`*.

The CGI Execution Environment

CGI scripts don't 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 installed 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.

Under Apache, you need to create a directory under your `$HOME/public_html` directory named `cgi−bin`. Both `$HOME/public_html` and its `cgi−bin` subdirectory should have "group" and "other" permissions set to allow both reading and executing for the Web server program 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.

```
% 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.

An Example HTML Form

CGI scripts are typically invoked from HTML pages, so for testing purposes, create an HTML form containing Listing 12−1, saved with the file name `simple.html`. 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, `simple.cgi` executable. When you view this page in your browser it should look something like the one shown in Figure 12−1.

Listing 12−1
An HTML form

```
<HTML><HEAD><title> An HTML Form Example </title></HEAD>
<BODY>
<h1> A <tt>cgi.icn</tt> Demonstration</h1>
<form method="GET"
      action="http://icon.cs.unlv.edu/cgi-bin/simple2.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>
<input type="checkbox" name="fortune">Fortune</input>
<input type="checkbox" name="grail">Grail</input><p>
4. Favorite Color:
<select name="color">
<option>Red
<option>Green
<option>Blue
<option selected>Don't Know (Aaagh!)
</select><p>
Comments:<br>
<textarea rows = 5 cols=60 name="comments"></textarea><p>
<input type="submit" value="Submit Data">
<input type="reset" value="Reset Form">
</BODY>
</HTML>
```

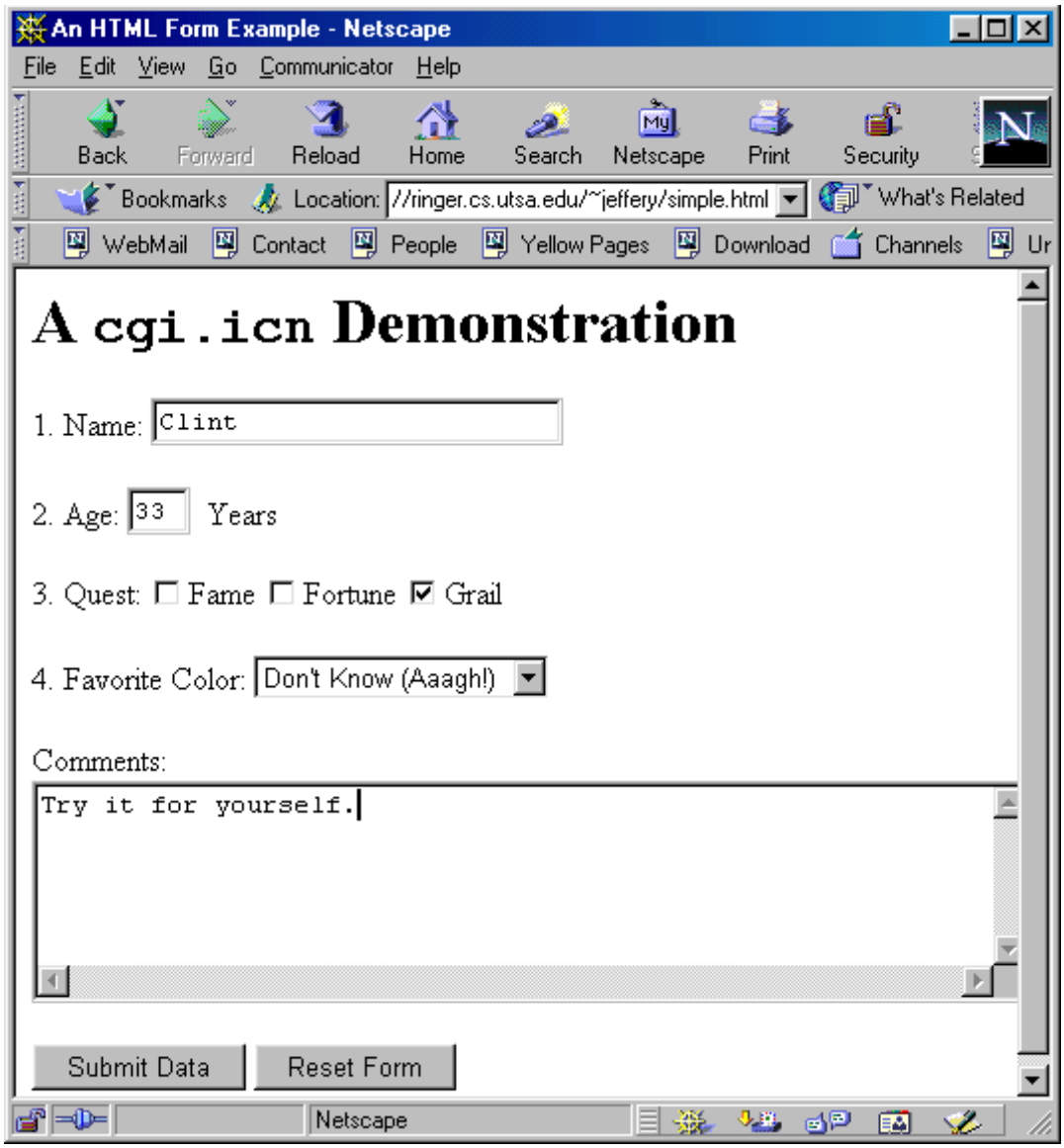

Figure 12–1:
An HTML Form Example

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 Icon source file, `simple.icn`, that you can copy from the CD–ROM disc. This program needs to be compiled with the command

```
icont –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.

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.

```
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.

Appform: An Online Scholarship Application

The next example, `appform.icn`, is a larger CGI script for an on−line scholarship application. The program has been in use for several years at the University of Texas at San Antonio. It follows a similar structure as the previous example, with a significant twist: the user input is formatted in LaTeX and a paper copy is printed for the convenience of the scholarship administrators. As a fail−safe measure, the CGI script also e−mails the application to the scholarship administrator. This is useful if the user typed something into the form that caused the LaTeX program to fail, or if the print failed for some reason.

The program is written as a single `cgimain()` procedure, which starts by processing each of the user input fields, escaping characters that are interpreted by LaTeX as commands. The program then opens a temporary file with a `.tex` extension, and writes a nicely typeset document containing the user's scholarship application information. You do not need to pay much attention to the LaTeX output, but notice that LaTeX uses two backslashes to generate a line break; these line breaks are everywhere. There are a couple of places where the form is printed differently depending on the user input. This allows the same CGI script to serve for both the regular scholarship and a summer program.

The code form Appform is showin in Listing 12−2. To run it you would have to adapt it to fit your environment; for example, as written it includes absolute paths to external applications (latex, dvips, lpr), and sends mail to user "jeffery". When running a CGI script it is important to realize you will run in a different directory, and with different user name and PATH environment, than your regular account. The program is run with whatever user permissions the system administrator has assigned the Web server process.

Listing 12−2
An online application form

```
############################################################################
###
#
#       File:     appform.icn
```

```
#
#       Subject:  CGI program to process scholarship applications
#
#       Author:  Clinton Jeffery
#
#       Date:     July 11, 1997
#
############################################################################
###
#
# This program processes a bunch of input fields defined in an
# on-line scholarship application at
# http://www.cs.utsa.edu/scholarships/mii.form.shtml
# and from them, generates a latex file which it typesets,
prints,
# and e-mails to the scholarship coordinator.
#
############################################################################
###

link cgi
link io

procedure cgimain()
  every k := key(cgi) do {
      s := ""
      cgi[k] ? {
          while s ||:= tab(upto('#%.&_{}"\\^~|<>-')) do {
              case c := move(1) of {
                  "\\": s ||:= ".\\setminus."
                  "^": s ||:= ".^{\\wedge}."
                  "~": s ||:= ".\\sim."
                  "|": s ||:= ".\\mid."
                  "<": s ||:= ".<."
                  ">": s ||:= ".>."
                  "-": s ||:= ".-."
                  default: {
                      s ||:= "\\"; s ||:= c
                  }
              }
          }
          s ||:= tab(0)
          }
      cgi[k] := s
  }
  fname := tempname("appform", ".tex", "/tmp")
  f := open(fname, "w") | stop("can't open /tmp/", fname)

  write("Generating typeset copy of application form...")

  write(f,"\\documentclass[11pt]{letter}")
  write(f,"\\pagestyle{empty}")
  write(f,"")
  write(f,"\\setlength{\\textwidth}{6.6in}")
  write(f,"\\setlength{\\textheight}{10in}")
  write(f,"\\setlength{\\topmargin}{-.3in}")
  write(f,"\\setlength{\\headsep}{0in}")
  write(f,"\\setlength{\\oddsidemargin}{0in}")
  write(f,"\\baselineskip 13pt")
  write(f,"\\begin{document}")
  write(f,"")
  write(f,"\\begin{center}")
  write(f,"{\\large\\bf")
  if (/(cgi["TESTSCORES"])) | trim(string(cgi["TESTSCORES"])) ==
"0"
```

```
  then
     write(f,"NSF CS Scholars and Mentors Program Application")
  else
     write(f,"NSF Summer Computer Science Institute Application")
  write(f,"}")
  write(f,"\\end{center}")
  write(f,"")
  write(f,"\\begin{tabular}{llll}")
  writes(f, "Name: & ", cgi["NAME"])
  writes(f," & Phone: & ", cgi["PHONE"])
  write(f," \\\\")
  write(f,"Address: & ", cgi["ADDRESS1"],
        " & Social Sec. Number: & ", cgi["SOC"], " \\\\")
  write(f,"\\ & ", cgi["ADDRESS2"], " & Gender (M/F): & ",
        cgi["GENDER"])
  write(f,"\\end{tabular}")
  write(f,"")
  write(f,"Semester hours completed: Overall ",
cgi["TOTALCREDITS"],
        " in Computer Science ", cgi["CSCREDITS"], "\\\\")
  write(f,"College GPA:  Overall ", cgi["COLLEGEGPA"],
        " in Computer Science courses ", cgi["CSGPA"],"\\\\")
  if (/(cgi["TESTSCORES"])) | trim(string(cgi["TESTSCORES"])) ==
"0"
  then
     write(f,"Are you interested in graduate studies? \\\\")
  else
     write(f,"Are you interested in a CS degree at UTSA?")
  write(f, if cgi["YES"] == "on" then "yes" else "", " ",
        if cgi["NO"] == "on" then "no" else "", " ",
        if cgi["MAYBE"] == "on" then "maybe" else "",
        "\\ \\\\")
  write(f,"Present Employer: ", cgi["EMPLOYER"], " \\\\")
  write(f,"Position: ", cgi["POSITION"], "Hours/week: ",
        cgi["HOURS"], " \\\\")
  write(f,"Will the scholarship enable you to quit your present")
  write(f,"job?  If not, how many hours will you be working?
\\\\")
  write(f, cgi["STILLWORKING"], "\\ \\\\")
  write(f,"Educational Background\\\\")
  write(f,"High School: name, dates attended, GPA,
graduated?\\\\")
  write(f, cgi["HIGH1"], "\\ \\\\")
  write(f, cgi["HIGH2"], "\\ ")
  if (/(cgi["TESTSCORES"])) | trim(string(cgi["TESTSCORES"])) ==
"0"
  then
     write(f," \\\\")
  else
     write(f,"Test Scores: ", cgi["TESTSCORES"], " \\\\")
  write(f,"For each college, list name, dates attended,",
        " hours completed, degrees awarded.")
  write(f,cgi["COLL1"], "\\ \\\\")
  write(f,cgi["COLL2"], "\\ \\\\")
  write(f,"\\ \\\\")
  write(f,"\\ \\\\")
  write(f,"Academic honors, scholarships, and fellowships\\\\")
  write(f,cgi["HONOR1"], "\\ \\\\")
  write(f,cgi["HONOR2"], "\\ \\\\")
  write(f,"\\ \\\\")
  write(f,"Extracurricular interests: \\hrulefill \\\\")
  write(f,cgi["EXTRA1"], "\\ \\\\")
  write(f,cgi["EXTRA2"], "\\ \\\\")
  write(f,"Professional organization
memberships:\\hrulefill\\\\")
```

```
  write(f,cgi["ORGS1"], "\\ \\\\")
  write(f,cgi["ORGS2"], "\\ \\\\")
  write(f,"Research interests and publications: \\hrulefill
\\\\")
  write(f,cgi["RESEARCH1"], "\\ \\\\")
  write(f,cgi["RESEARCH2"], "\\ \\\\")
  write(f,"Military Service or Draft Status: \\hrulefill \\\\")
  write(f,cgi["MIL1"], "\\ \\\\")
  write(f,cgi["MIL2"], "\\ \\\\")
  write(f,"Name(s) of at least one person you have asked to
write")
  write(f,"an academic reference letter.\\\\")
  write(f,"Name \\hfill Address \\hfill Relationship \\\\")
  write(f,cgi["REF1"], "\\ ", cgi["REFADD1"], "\\ ",
        cgi["REFREL1"],"\\\\")
  write(f,cgi["REF2"], "\\ ", cgi["REFADD2"], "\\ ",
        cgi["REFREL2"],"\\\\")
  write(f,"")
  write(f,"Please write a {\\em short\\/} statement of purpose,")
  write(f,"including information about your background,")
  write(f,"major and career interests, and professional goals.")
  write(f,"")
  write(f,"I certify that information provided on this
application")
  write(f,"is correct and complete to my knowledge. \\\\ \\ ")
  write(f,"")
  write(f,"\\noindent Signature: \\rule{3.5in}{.01in}")
  write(f," Date: \\hrulefill")
  write(f,"")
  write(f, "\\pagebreak")
  write(f,"")
  write(f, cgi["INFO"])
  write(f,"\\end{document}")
  close(f)
  write("Mailing form to program director...")
  system("cd /tmp; mail jeffery < " || fname)
  write("Typesetting and Printing hard copy...")
  system("cd /tmp; /usr/local/bin/latex " || fname ||
         "    >/dev/null 2>/dev/null; " ||
         " /usr/local/bin/dvips -o " || fname || ".ps " || fname
||
         "    >/dev/null 2>/dev/null; " ||
         " /usr/ucb/lpr " || fname || ".ps; rm " || fname ||
".*")
  cgiEcho("Thank you for applying, ", cgi["NAME"])
  cgiEcho("Your application has been submitted to user jeffery")
end
```

Summary

Writing CGI scripts in Icon is quite 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 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 13: 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 take control of and extend their own system. This chapter presents the techniques used to write several simple 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.

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. You also need the POSIX definitions to find information about files. Here is the start of a file−search application.

```
#
# search.icn
#
# Search for files whose names match a pattern (regular
expression)
#
# Usage: ./search <pattern> [dirs]

link regexp
```

The application should start by processing the command−line arguments. There needs to 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:

```
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)
end
```

The search algorithm is a depth−first search. In each directory, check the type of each file. If you find a directory, immediately 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.

```
procedure findfile(dir, pattern)
   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 links, the directory hierarchy is a tree so you don't need to check for cycles.

```
   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)
   }
end
```

This is a very simple demonstration of some systems programming techniques for Unicon. You will see this sort of depth−first traversal of the filesystem again, in the section on filesystem backups later in this chapter.

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)
   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)
      }
   }
end
```

The table named `lengths` is a mapping from lengths to filenames. 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 table will contain the following:

```
   lengths[1] === [ "A" ]
   lengths[2] === [ "B", "D" ]
   lengths[5] === [ "C" ]
```

The main program scans all the directories, and for each list, compares all the files in it to each other:

```
global lengths
procedure main()
   lengths := table()
   scan("/")
   every l := !lengths do {
      if *l = 1 then next
      find_dups(l)
   }
end
```

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

```
   lengths[1] === "A"
   lengths[2] === [ "B", "D" ]
   lengths[5] === "C"
```

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:

```
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.

```
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 a hashing technique, 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.

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 `compare()` to perform the actual comparisons.

```
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
```

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.

> **Tip**
>
> By clever use of hashing, you may be able to get away with reading each file just once.

Listing 13−1 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 13−1
A program for finding duplicate files.

```
#
# duplicate.icn
#
# Find files in the filesystem that are identical

$include "posix.icn"

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)
      }
end

# Scan all the directories and add files to the length map −
# the global table "lengths"
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 just 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)
   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 your i/o speed compared to CPU
# speed and memory.
procedure compare(file1, file2)
   static maxline
   initial maxline := 1000

   # are f1 and f2 are identical?
   f1 := open(file1) | {
      write(&errout, "Couldn't open ", file1, "; ",
            sys_errstr(&errno))
      fail
      }
   f2 := open(file2) | {
      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
         fail
      }
   return ""
end
```

User File Quotas

Many computing platforms offer a filesystem quota facility, where each user can only use so much of the disk to store files. The system will enforce this by not allowing any 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. Most of the application is portable, but at present the e−mail message delivery only works on UNIX.

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. First, declare a few global variables that will hold the strings that are to be used in the messages that are sent:

```
global complaint_part1, complaint_part2, summary_header, mailer

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"
   mailer := "mail"
   init_strings()
```

Calculating Disk Usage

The next step is to read in the database of users, and for every directory mentioned, calculate the disk usage. On UNIX systems you can just run `du` (the UNIX utility 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 would it be to write one? The `du` command is actually very straightforward to write, using the techniques presented in the previous two programming examples.

```
   L := read_db(db)
   owners := L[1]
   quotas := L[2]
   daysover := L[3]
   usages := table(0)
   over := table()
   every dir := key(owners) do {
      p := open("du −s " || dir, "p")
      usages[dir] := (read(p) ? tab(upto('\t')))
      user := owners[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.

```
      if usages[dir] > quotas[dir] then {
         /over[user] := []
         daysover[dir] +:= 1
         l := [dir, usages[dir], quotas[dir], daysover[dir]]
         push(over[user], l)
         }
      else {
         daysover[dir] := 0
         }
      }
   every user := key(over) do {
      complain(user, over[user])
      }
```

Finally, the program saves the database and sends a summary of the over−quota directories to the administrator.

```
   write_db(db, owners, quotas, daysover)
   send_summary(administrator, owners, quotas, daysover, usages)
end
```

Sending a Mail Message

The code that actually transmits a mail message presents another portability challenge. 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. On a UNIX system, you can send this message by simply opening a pipe to the `mail` program, and writing your message. The `mail` program takes a user id on its command line; the `−s` option specifies a message subject.

```
procedure complain(user, L)
   msg := "Dear " || user || complaint_part1
   every l := !L do {
      msg ||:= "\t" || l[1] || "\t" || l[2] || "\t" ||
         l[3] || "\t" || l[4] || "\n"
   }
   msg ||:= complaint_part2
   m := open(mailer || " −s 'Over Quota' " || user, "pw")
   write(m, msg)
   close(m)
end
```

If your system does not have a UNIX−compatible `mail` program, how easy would it be to write one? It is not at all obvious how to write such a program in Icon. On Windows, you might write such a mail client in C using MAPI. To write `mail` in Unicon you would have to connect to a mail server directly using a mail protocol such as SMTP. Another option is to extend the Unicon language to support mail as a built−in feature. This has been done for many flavors of UNIX as part of the Messaging Facilities and is described in Appendix C. It is expected that the Messaging Facilities will be ported to all Unicon platforms and become a regular part of the language, solving the portability problem inherent in this program example.

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(mailer || " −s 'Quota Summary' " || admin, "pw")
   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
```

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)
   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 multiple quota lines occur for a directory, the tables must be updated appropriately. The semantics of the tables require varying approaches. The owners table writes a warning message if quota lines with different owners for the same directory are found, but otherwise the owners table is unaffected by multiple entries. The actual quotas table allows multiple quota lines for a directory; in which case the quotas are added together. The daysover table retains the maximum value any quota line is overdue for a directory.

```
         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)
            next
         }
         dir := tab(upto(' \t'))
         write(db_new, dir, "\t", owners[dir], "\t", quotas[dir],
               "\t", daysover[dir])
      }
   }
   close(db_old)
```

```
   close(db_new)
   rename(db, db || ".bak")
   rename(new_db, db)
end
```

Lastly, procedure `init_strings()` initialises 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
```

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"

$include "posix.icn"

procedure main(L)
   if L[1] == "-a" then {
      flags := "a"; pop(L)
      }
   else
      flags := "w"
```

Now the program must find a pty to use. A standard solution is to go down the list of all pty device names in sequence until an `open()` succeeds; then the `capture()` function is called, which performs 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!
            capture(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 echo to the user, and also saved to the script file.

```
procedure capture(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
```

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, so that 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 floppy disks and Zip® disks. It does not work on many 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. This might include temporary directories like "`/tmp`" on UNIX systems. One directory that definitely 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.

```
#
# 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 floppy or Zip drive.
#
# Example:
#    backup -n3 -o/mnt/zip /home/bob

$include "posix.icn"

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)

end
```

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.

```
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.ctime > 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.

```
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 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

   return prev
end
```

Finally, the program saves the database of dates. It fetches the current time to save with the current level, and it also deletes all higher-numbered backups from the database.

```
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
```

Summary

Writing utilities and system administration tools is easy in Unicon. These programs rely especially heavily on the system facilities described in chapter 5. While we do not advocate using Unicon in every case, you may find it to be 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.

Chapter 14: Internet Programs

The Internet is easily the most important development in the past ten years of computing. Because it is ubiquitous, programmers should be able to take it for granted, and 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

The Client−Server Model

The Internet allows you to distribute applications using any topology you wish, 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 is easy, in fact, 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 a bit 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.

Writing a server program is a little bit harder. 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.

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 connected 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 you see the server program, 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 three lines consisting of 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. The Sesrit program presented in Chapter 16 is an example program that uses this client procedure. 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, game, "\n", userid, "\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 `scored` must handle the request. The `main()` procedure given below allows the user to specify a port, or uses a default port if none is supplied.

```
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 `score_result()` does all the real work of the server, and its implementation is of architectural significance. If any delay is possible in handling a request, the server will be unable to handle other simultaneous client requests. For this reason, many server programs immediately spawn a separate process to handle each request. You could do that with `system()`, as illustrated in Chapter 5, but for `scored` this is overkill. The server will handle each request almost instantaneously itself.

The `score_result()` procedure is of additional interest. It 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.

```
procedure score_result(net)
   static t
   initial t := table()

   game := read(net) | { write(net,"no game?"); fail }
   owner := read(net) | { write(net,"no owner?"); fail }
   score := read(net) | { 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]
```

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 {
      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[name]` 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
```

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")
```

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 important thing to remember when handling input from multiple sources is that you absolutely cannot block for I/O. This means when handling network connections, you should never use `read()`, you should only use `reads()`. For windows you should 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.

```
   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 `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 `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.

```
procedure handle_char(w, c)
   # Current horiz. position for wach half of the window
   static xpos
   initial xpos := [margin, margin]

   if c == "\^d" then fail     # EOF

   # Find the half of the window to use
   y_offset := (w − 1) * wheight/2

   if c == ("\r"|"\n") | xpos[w] > wwidth then {
      ScrollUp(y_offset+1, wheight/2−1)
      xpos[w] := margin
      }
   if c == ("\r"|"\n") then return
   #handles backspacing on the current line
   if c =="\b" then {
      if xpos[w] >= margin + cwidth then {
         EraseArea(xpos[w] − cwidth, y_offset+1 + wheight/2−1 −
vs,
```

```
                  cwidth, vs)
            xpos[w] −:= cwidth
            return
            }
         }
   DrawString(xpos[w], 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 `CopyArea()` is used to move the existing contents up one line, after which `EraseArea()` clears the line at the bottom.

```
procedure ScrollUp(vpos, h)
   CopyArea(0, vpos + vs, wwidth, h−vs, 0, vpos)
   EraseArea(0, vpos + h − vs, wwidth, vs)
end
```

The window is initialized with a call to the library procedure, `WOpen()`. The `WOpen()` procedure 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 `vs`, which gives the vertical space in pixels between lines.

```
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 `give_up()` writes a message and exits the program, if no network connection is established. If `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 `stop()`.

```
procedure give_up(host, user)
   stop("no connection to ", (\user || "@") | "", host)
end
```

What enhancements would make `italk` more interesting? A first and most obvious extension would be to use a standard network protocol, such as that of UNIX talk, so that `italk` could communicate with other users that don't have `italk`. UNIX `talk` also offers a more robust connection and authentication model (although you are still dependent on the administrator of a remote machine to guarantee that its `talkd` server is well behaved). Another feature of UNIX `talk` is support for multiple simultaneously connected users.

One particularly neat extension you might implement is support for graphics, turning `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!

Summary

Writing Internet programs can be easy and fun. 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 15: 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.

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 simply examine the fitness of this one individual with an exhaustive search:

```
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:

```
maxi := 2^6
population := [?maxi, ?maxi, ?maxi, ?maxi, ?maxi, ?maxi]
every i := 1 to 100 do {
   worst := 1
   every i := 1 to 5 do
      if population[worst] <= population[i] then
         worst := i
   population[i] := ?maxi
   }
```

Before modeling mating and mutation, you'll have to create a more detailed representation of the internals of an individual.

GA Operations

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:

```
population := list(6)
population[1] := "010111"
population[2] := "000101"
population[3] := "111101"
population[4] := "111011"
population[5] := "111110"
population[6] := "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:

```
population[1] := "010111" ¢ 23 winner
   population[6] := "010110" ¢ 22
```

```
population[3] := "111101" ¢ 61 winner
population[4] := "111011" ¢ 59

population[5] := "111110" ¢ 62 winner
population[2] := "000101" ¢ 5
```

The second round of selections is listed here:

```
population[5] := "111110" ¢ 62 winner
population[6] := "010110" ¢ 22

population[2] := "000101" ¢ 5
population[3] := "111101" ¢ 61 winner

population[4] := "111011" ¢ 59 winner
population[1] := "010111" ¢ 23
```

The end result of tournament selection is listed here:

```
next_gen[1] := population[1] := "010111" ¢ 23
next_gen[2] := population[3] := "111101" ¢ 61
next_gen[3] := population[5] := "111110" ¢ 62
next_gen[4] := population[5] := "111110" ¢ 62
next_gen[5] := population[3] := "111101" ¢ 61
next_gen[6] := population[4] := "111011" ¢ 59
```

Notice how there are two copies of `population[3]` and `population[5]` in `next_gen`. On the other hand, there are no copies of `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 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.

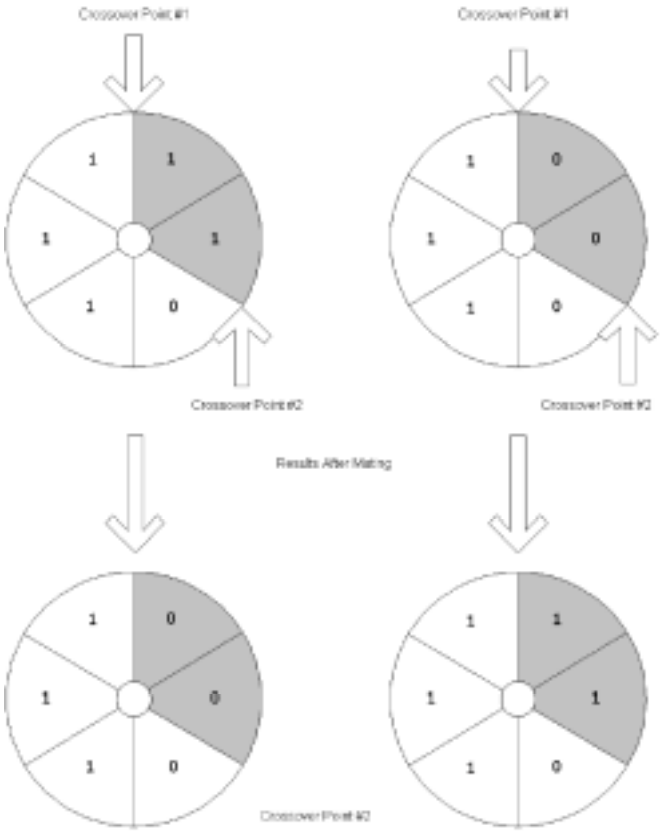

Figure 15−1:
Two−point crossover.

The code for two−point crossover is presented below. The `?(lchrom+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 `parent1` and `parent2` within the range from `a` to `b`.

```
a := ?(lchrom+1)
while ((b := ?(lchrom+1)) = a)
if a > b then a :=: b
ncross +:= 1
child1 := parent1[1:a] || parent2[a:b] || parent1[b:0]
child2 := parent2[1:a] || parent1[a:b] || parent2[b:0]
```

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.

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 is still 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 only one of many implementation choices you have to make. There are all sorts of choices you can 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/*l*, 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.

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 CD−ROM that accompanies this book.

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 ga_eng

Class `ga_eng` implements the engine. It provides the following two public methods: `evolve()` and `set_params()`, as well as eight private methods:

```
method set_params(fitness_func_in, popsize_in, lchrom_in,
                  pcross_in, pmutation_in, log_file_in)
method evolve()

method tselect()
method crossover2(parent1, parent2)
method generation()
method stats(Pop)
method report(Pop)
method random_chrom()
method initpop()
method init()
```

The `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 `ga_eng()` takes the same input parameters as `set_params()` but it also re−initializes the engine by creating a new

population from scratch. The `evolve()` method moves the population from one generation to the next.

The `tselect()` method operates on the whole population by performing tournament selection. The `crossover2()` method takes two individuals and returns two new individuals in a list after doing two−point crossover. The `stats()` method collects statistics on the given populations, and `report()` writes the results out to the log file if it is not set to `&null`. The `init()` and `initpop()` methods initialize the GA.

You might be asking what is a population of *individuals*? The key piece of information about an individual is 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:

```
record individual(chrom, fitness, parent1, parent2, xsite)
```

This stores the fitness values, the index of two parents, and the crossover sites from when the parents were mated.

The class `ga_eng` makes use of the following instance variables:

* `oldpop`, `newpop` – two populations; selection goes from `oldpop` into `newpop`
* `popsize`, `lchrom` – the population size and the length of the chromosomes
* `gen`, `maxgen` – current and max generation number
* `pcross`, `pmutation` – the probability of crossover and mutation
* `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 prevent us from discussing every method of `ga_eng`, but `evolve()` is a method that is called directly from user code, and defines the basic architecture of the engine. The `evolve()` method initializes the population the first time the method is invoked. Unicon supports this with the `initial {}` section of a procedure or method. After initialization and in all 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:

```
method evolve()
   initial {
      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. The first 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 inside of a loop, on pairs of individuals. The `crossover2()` method does the mating, and it encapsulates the relatively low−level operation of mutation. The last high−level operation that a GA does is to call the user supplied `fitness_func` to assign a fitness value to each individual in the new population. The GA needs to keep 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 15−1 shows the code for the method`generation()`:

Listing 15−1

A method for producing a new generation.

```
method generation()
   local j, mate1, mate2, jcross, kids, x,
       fitness1, fitness2, selected

   newpop    := list(popsize)
   nmutation := ncross := 0
   selected  := tselect()

   j := 1
   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 very easy to create a buggy GA that works so well that the bugs go undetected. Incidentally, masking of bugs by robust algorithms is not unique to GA. It occurs in many numerical algorithms. To prevent this masking, 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 15−2 shows all the code needed is to use the GA engine for a simple test problem.

Listing 15−2

Using ga_eng() for a simple test problem

```
procedure decode(chrom)
   local i, s
   i := 0
   every s := !chrom do
      if s == "1" then i +:= 1
   return i
end

# a simple test of the engine, fitness is based on number of
# 1 bits in chrom
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 15−3 is a fragment of the log file that is generated. The complete log file has over twelve thousand lines.

Listing 15−3

A trace of the GA for a simple test problem

```
Log_File for Genetic Algorithm (GA)

 Population size          = 100
 Chromosome length        = 20
 Maximum # of generations =
 Crossover probability    = 0.99
 Mutation probability     = 0.05

 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

                              Generation   0
```

```
     #    parents       xsite   chromo    fitness
------------------------------------------------------------
       1) (  0,   0)       0    0001000111000010000 10  5.00e-1
       2) (  0,   0)       0    1101110101101000110 0  5.00e-1
       3) (  0,   0)       0    0110110000011010011 1  5.00e-1
.........
     100) (  0,   0)       0    1101110011001100101 0  5.00e-1
------------------------------------------------------------

Statistics:
min =  5.00000000e-1  avg =  5.00000000e-1  max =  5.00000000e-1

 no. of mutations        = 0
 no. of crossovers       = 0
 location of best chromo = 1
------------------------------------------------------------
dateline = Wednesday, January 27, 1999  10:37 pm
.................
Population Report
                              Generation 100

     #    parents       xsite   chromo    fitness
------------------------------------------------------------
       1) (  1,   2)   15:21    0001101101100001010 0  9.19e0
       2) (  1,   2)   15:21    1101100101111110001 0  1.08e1
       3) ( 90,  72)   14:18    1001111111111111111 1  1.78e1
.....
     100) ( 57,  27)   13:15    1111101011011011111 0  1.78e1
------------------------------------------------------------

Statistics:
min =  9.19999999e0   avg =  1.77200000e1   max =  1.99500000e1

 no. of mutations        = 111
 no. of crossovers       = 51
 location of best chromo = 46
------------------------------------------------------------
dateline = Wednesday, January 27, 1999  10:37 pm
best location => 46
best fitness  => 19.95
```

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 to display 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. The wheel has the drawback that 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 the numeric codes for the color by hand. Through trial and error, by examining a large number of colors you can select the right color. 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 index with *`i`*. where *`i`* runs from 1 to 16. The value of the scrollbar is inverted because 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 {
   f := VGetState(vidgets["sbar"||i])
   ga.newpop[i].fitness := 1 - f
}
```

Gradually the population evolves from a set of random colors to a set of colors that all look alike, with ever so slight variations on the one you had in your mind − 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 15−2 is a screen snapshot of `cb`. The higher the user slides the scrollbar tab, the more fit the color. There are three resolutions of color 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 mixing of foreground and background colors. They come in three different resolutions: 6x6, 8x8, and 10x10. In text mode, the foreground color is displayed as text on top of the background.

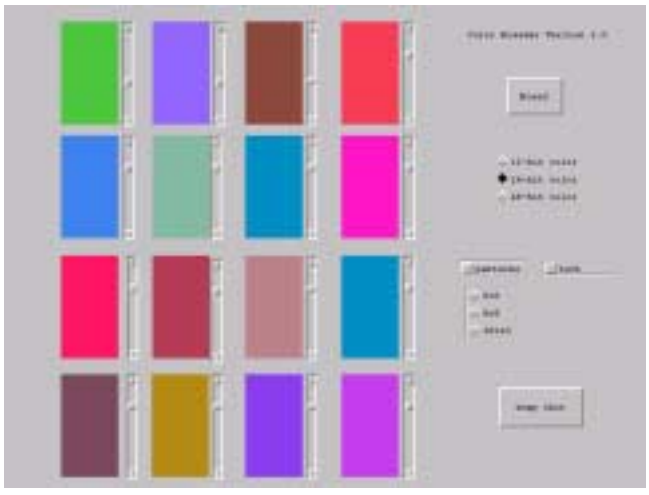

Figure 15−2:
`cb` − a genetic algorithm for picking colors.

Breeding Textures

The user can also turn on bi−level patterns. A bi−level pattern is composed of three parts: a foreground color, a background color, and a square bit pattern. The pattern is used as a tile to fill a whole area. Where the pattern has a one, the foreground color is used, and where it has a zero, the background color is used. The bit patterns come in three different resolutions: 8x8, 10x10, and 12x12. With a low resolution, it is hard to find interesting patterns. With a high resolution, most of the patterns look like a random mixture and are hard to evolve into something more structured. In Icon, bi−level image specifications have the following form: *width, #data*. The data is a series of hex digits that specify the row values from top to bottom. Figure 15−3 shows `cb` in pattern mode.

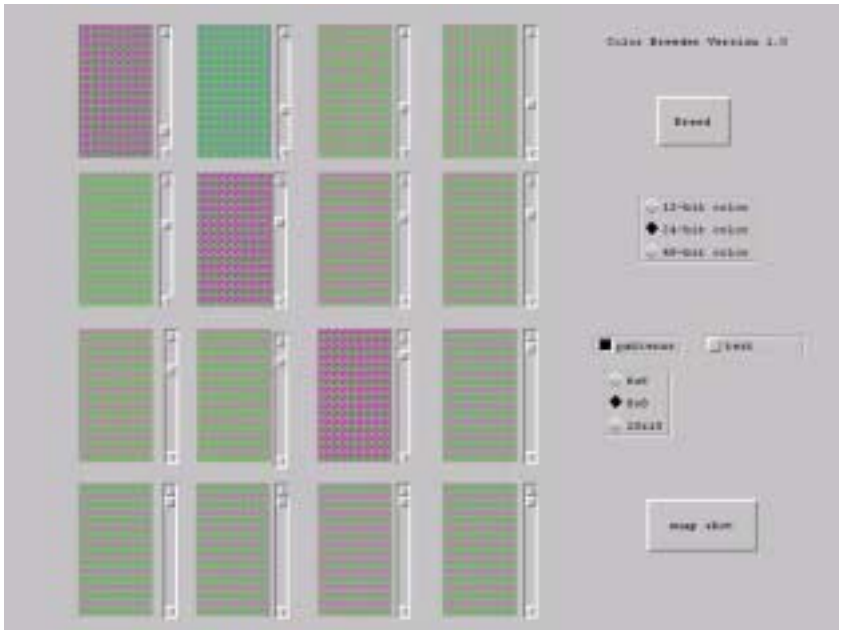

Figure 15−3:
`cb` in pattern mode

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.

Picking colors for text displays

Perhaps the most fun use of `cb` is to pick colors for your Web pages. Figure 15−4 is a snapshot of `cb` in text mode:

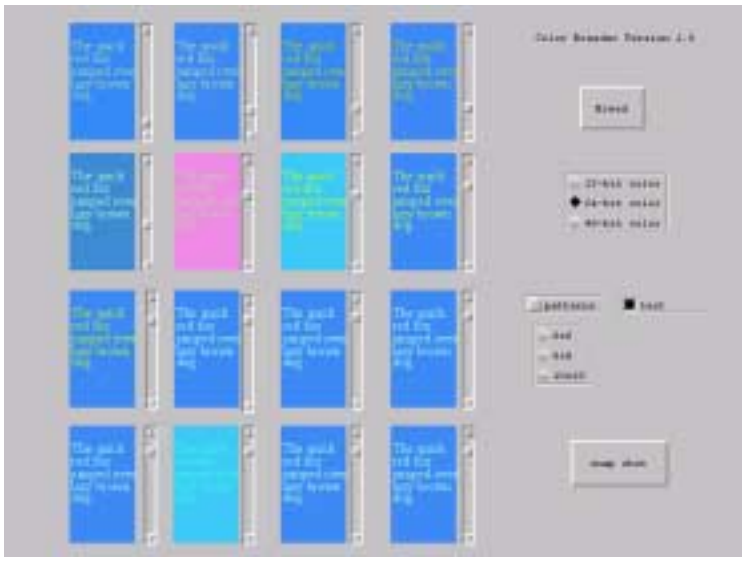

Figure 15−4:
cb in text mode.

In text mode with 24−bit color resolution, the user can right click on a color region to generate a sample HTML document such as the one shown in Figure 15−5 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.

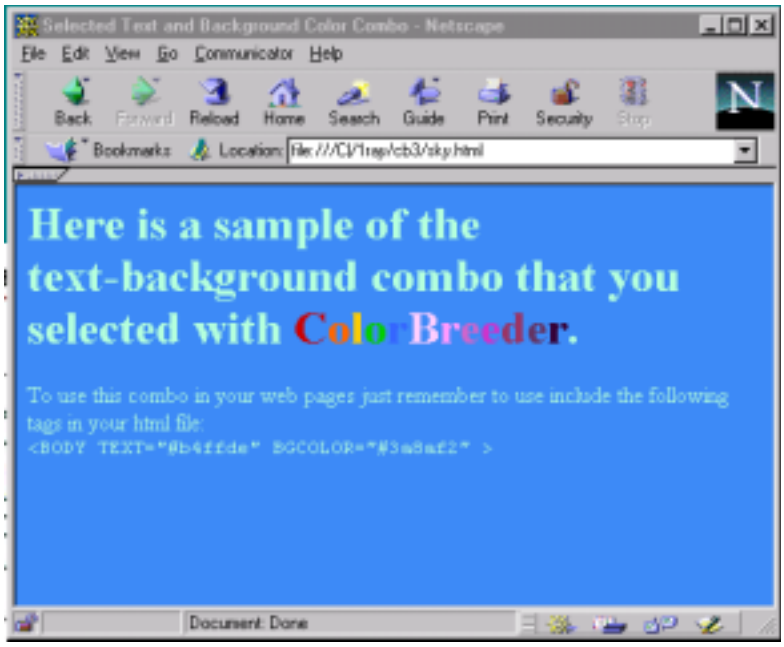

Figure 15−5:
A sample HTML page generated by cb

Summary

GAs form a branch of evolutionary computation. They make up a very general−purpose optimization technique. The three main GA operations are selection, mating, and mutation. Object−oriented techniques can be used to create 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 16: Games

Computer games are an extremely active industry segment that concerns both professionals and hobbyists. Icon is an excellent language for writing simple games and for rapidly developing prototypes of complex games. Icon will never be the best language for the hardware−dependent, low−level graphics programming required for cutting−edge video games, but for most kinds of games it is ideal. To fully understand some of the graphics facilities used in this chapter you may wish to consult the book Graphics Programming in Icon. This chapter shows you:

- How to animate objects within a graphics screen.
- How to represent on−screen objects with internal data structures.
- Steps to take in an object−oriented design for a complex game.
- An example of a moderately sophisticated custom user interface design.

Sesrit

Sesrit is a game by David Rice based on the classical game of Tetris. A free software program called xtetris inspires Sesrit's look and feel. Sesrit is written in about five hundred lines of Icon.

In Sesrit, like Tetris, pieces of varying shape fall from the top of a rectangular play area. Sesrit, however, uses randomly generated shapes, making it more difficult than tetris, which uses a small, fixed set of shapes for its pieces. The object of the game is to keep the play area clear so that there is room for more objects to fall. The play area is a grid of cells, ten cells wide and thirty cells high. Sesrit pieces fall at a rate that begins slowly, and increases in speed as play progresses.

A piece stops falling when it reaches the bottom row, or when one of its cells has another piece directly beneath it, preventing its fall. When a piece stops falling, a new randomly selected piece is created and starts to fall from the top of the play area. If the cells in a given row completely fill, they dissolve, and all cells above that row drop down.

The user moves the falling piece left or right by pressing the left or right arrow keys. The user may rotate the piece clockwise or counterclockwise by pressing the up arrow or down arrow, respectively. The spacebar causes the falling object to immediately drop as far as it can. Figure 16−1 shows a sample screen from *Sesrit*.

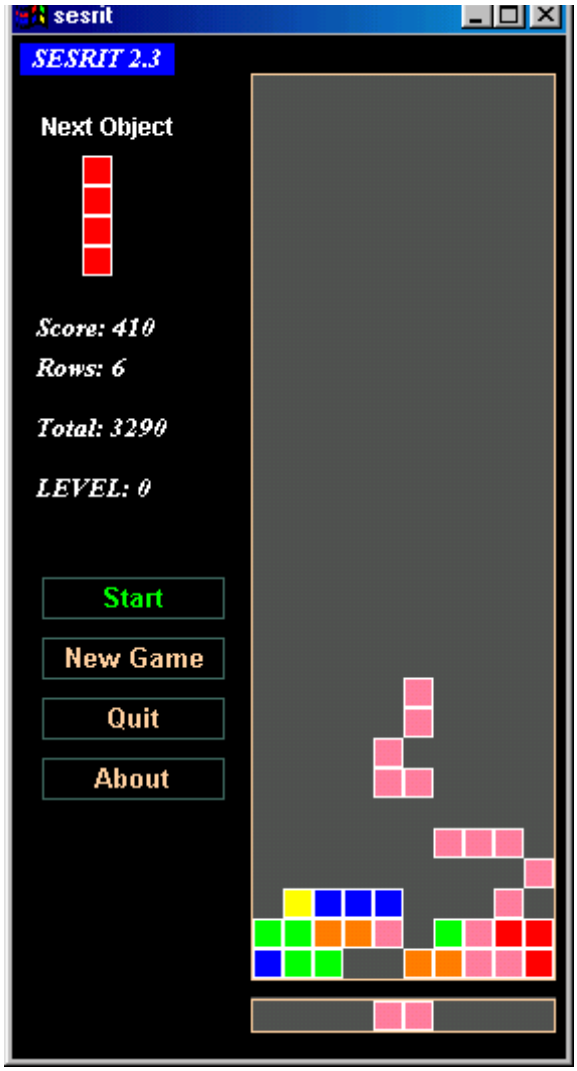

Figure 16−1:
An example screen image from the Sesrit game.

This section presents the key elements of *Sesrit*'s design and implementation. A few of the more mundane procedures are not described. The complete *Sesrit* source code is on the CD−ROM that accompanies this book. Like most games with a graphical user interface, *Sesrit* starts with link declarations that give it access to graphics library procedures and to a function that randomizes program behavior on each run. *Sesrit* also links in the Internet high score client procedure from Chapter 14. Including the file `"keysyms.icn"` introduces defined symbols for special keyboard keys such as the arrow keys used in *Sesrit*.

```
link graphics
link random
link highscor
$include "keysyms.icn"
```

Sesrit has a number of global variables that are used to maintain its internal representation of the game state and screen contents. The global variable `L` represents the actual playing area contents as a list of lists of cells, where each cell is represented by the string name for the color of that cell. For example, `L[12, 7]` could have the value `"red"`.

Several other global variables are important. A list of two−element lists containing (x,y) coordinates represents the current piece (variable `activecells`) and the next piece (variable `nextpiece`). Other global variables maintain details such as the rate at which the pieces fall, and the current score. The global table named `colors` holds a mapping from colors' string names to window bindings whose foreground colors are set to that color.

```
global activecells, activecellcolor, nextpiece, nextcolor, L,
       colors, score, numrows, level, delaytime, pieceposition,
       button_status, game_status, tot_score
```

You can draw a cell such as `L[12,7]` with a call to `drawcell(x, y, L[12, 7])`. *Sesrit*'s procedure `drawcell(x,y,color)` fills a rectangle of the appropriate color and then (for nonempty cells) draws a white border around it.

```
procedure drawcell(x,y,color)
   FillRectangle(colors[color],x,y,15,15)
   if color ~=== "black" then
      DrawRectangle(colors["white"], x, y, 14, 14)
end
```

The `main()` procedure of *Sesrit* initializes the graphics and state variables and then executes a loop that allows the user to play one game on each iteration. After each game the user has the option of playing again, or quitting. For each game, the main action is accomplished by a call to procedure `game_loop()`.

```
procedure main()
   init()
   repeat
      if buttons(15, 105, 270, 290, ["Start", "green", 45, 285],
                  ["Pause", "red", 40, 285]) == "done" then break
   repeat {
      game_loop()
      init()
      }
end
```

Sesrit performs initialization with a procedure named `init()`. The first time `init()` is called, it must do a bit more work than in subsequent calls, so it has an initial section, which starts by opening a window and creating the color table. It then calls `randomize()`, from the Icon Program Library. The `randomize()` procedure is used in many games. Icon's random number generator normally uses the same sequence each time it executes, which is very helpful for debugging, but not a good feature in games that are supposed to be unpredictable. The `randomize()` procedure sets the random number seed based on the current time, so every game is different.

```
procedure init()
   initial {
      &window := WOpen("label=sesrit","size=276,510", "posx=20")
      colors := table(&window)
      every c := ("blue"|"yellow"|"cyan"|"green"|"red"|"white"|
                  "red−yellow" | "purple−magenta") do
         colors[c] := Clone("fg=" || c)
      colors["black"] := Clone("fg=dark vivid gray")
      randomize()
      }
```

The rest of the `init()` procedure consists of drawing the window's starting contents and initializing global variables. Most of that is not worth presenting here, but it is worth showing how *Sesrit*'s "playing field" (global variable `L`) and first two objects are initialized. `L` is a list of 30 lists of 10 elements that should all start as `"black"`. The

`list(n,x)` procedure creates a list of `n` elements, all with the initial value `x`. You cannot just initialize the variable `L` to `list(30, list(10, "black"))` because that would create a list of 30 references to a single list of 10 cells. The inner call to `list()` would only get called once. Instead, each of `L`'s thirty rows is initialized with a different list of 10 cells.

```
   ...
   L := list(30)
   every !L := list(10, "black")
   newobject()
   activecells := copy(nextpiece)
   activecellcolor := copy(nextcolor)
   every point := !activecells do
      L[point[1], point[2]] := activecellcolor
   newobject()
end
```

With the window and variables initialized, the main task of the game is to repeat a sequence of steps in which the current piece falls one row each step, until the game is over. Like many other games, this infinite loop starts with a check for user input, and since several events could be waiting, the check for user input is itself a loop that terminates when the window's list of pending events (returned by `Pending()`) is empty. A case expression performs the appropriate response to each type of user input. Notice how ordinary keyboard characters are returned as simple one−letter strings, while special keys such as the arrows have defined symbols. Mouse event codes have Icon keyword constants such as `&lpress` to represent their value. The `&lpress` constant indicates a left mouse key press. When `Event()` returns it assigns keywords `&x` and `&y` to the mouse location, so it is common to see code that checks these keywords' values while processing an input event. Several other keywords (`&control, &meta, &shift`) are also set to indicate the state of special keys (Control, Alt, and Shift on most keyboards). These keywords fail if the special key was not pressed at the time of the event..

```
procedure game_loop()
   game_status := 1
   repeat {
      while *Pending() > 0 do {
         case Event() of {
            Key_Left  : move_piece(−1, 0)
            Key_Right : move_piece(1, 0)
            Key_Down  : rotate_piece(1, −1)
            Key_Up    : rotate_piece(−1, 1)
            " "       : while move_piece(0, 1) # drop to bottom
            "q"       : if &meta then exit()
            "a"       : if &meta then about_itetris()
            "p"       : if (&meta & game_status = 1) then pause()
            "n"       : if &meta then return
            &lpress   : {
               if 15 <= &x <= 105 then {
                  if 270 <= &y <= 290 then pause()
                     else if 300 <= &y <= 320 then return
                        else if 360 <= &y <= 380 then
                           about_sesrit()
                  }
               }
            &lrelease :
               if ((15 <= &x <= 105) & (330 <= &y <= 350)) then
                  exit()
            } # end case
         } # end while user input is pending
```

Once user input has been handled, the piece falls one row, if it can. If the object could not fall, it is time to either bring the next object into play, or the game is over because the object is still at the top of the screen. The `game_over()` procedure is not shown in detail; it marks the occasion by drawing random colors over the entire screen from bottom to top and top to bottom and then asks whether the user wishes to play again.

```
      if not move_piece(0, 1) then {
         if (!activecells)[1] < 2 then { # top of screen
            game_over()
            return
            }
```

In the more common occurrence that the object could not fall but the game was not over, procedure `scanrows()` is called to check whether any of the rows are filled and can be destroyed. The next piece replaces the active cell variables, and procedure `newobject()` generates a new next piece. The score is updated for each new object, and the current and next pieces are drawn on the display.

```
            while get(Pending())
            scanrows()
            Fg("black")
            drawstat(score, , , tot_score)
            score +:= 5
            tot_score +:= 5
            Fg("white")
            drawstat(score, , , tot_score)
            activecells := copy(nextpiece)
            activecellcolor := copy(nextcolor)
            every point := !activecells do
               L[point[1], point[2]] := activecellcolor
            newobject()
            EraseArea(120,481,150,15)
            Bg("black")
            every cell := !activecells do {
               EraseArea(-40 + (cell[2]-1)*15, 60 + (cell[1]-
1)*15,
                         15, 15)
               drawcell(120 + (cell[2]-1)*15, 481,
activecellcolor)
               }
            every cell := !nextpiece do
               drawcell(-40 + (cell[2]-1)*15,
                   60 + (cell[1]-1)*15, nextcolor)
            }
```

Each step is completed by a call to `WSync()` to flush graphics output, followed by a delay period to allow the user to react. The `WSync()` procedure is only needed on window systems that buffer output for performance reasons, such as the X Window System. The delay time becomes smaller and smaller as the game progresses, making it harder and harder for the user to move the falling pieces into position.

```
      WSync()
      delay(delaytime)
      }
end
```

Procedure `newobject()` generates a new object, which will be displayed beside the game area until the current object stops falling. The object is stored in global variable `nextpiece`, and its screen color is given in variable `nextcolor`. Objects are represented as a list of (row,column) pairs where the pairs are two-element lists.

One quarter of the objects (?4 = 1) are of a random shape; the remainder are taken from the set of four−cell shapes found in *xtetris*, the free version of Tetris for X Window System platforms. Random shapes are obtained by starting with a single cell, and adding cells in a loop. Each time through the loop there is a 25% chance that the loop will terminate and the object is complete. The other 75% of the time (?4 < 4), a cell is added to the object, adjacent in a random direction from the last cell. The expression ?3−2 gives a random value of 1, 0, or −1. This expression is added to each of the x and y coordinates of the last cell to pick the next cell.

```
procedure newobject()   pieceposition := 1
   if ? 4 = 1 then {
      nextcolor := "pink"
      nextpiece := [[2,6]]
      while ?4 < 4 do {
         x := copy(nextpiece[−1])
         x[1] +:= ?3 − 2
         x[2] +:= ?3 − 2
         if x[1] = ((y := !nextpiece)[1]) & x[2] = y[2] then next
         put(nextpiece, x)
         }
```

There is a bunch of sanity checking that is needed for random objects, given below. If the object is so big that it won't fit in the next piece area beside the playing field, it is filtered out. In addition, we need to center the object horizontally. The subtlest task is to move the "center" cell of the random object (if it has one) to the front of the list, so the object rotates nicely. To do all this, the code first computes the boundaries (min and max) of the random object's row and column values. The solution for finding a center cell, checking each cell to see if it is at row `(miny + maxy)/2`, column `(minx + maxx)/2` is pretty suboptimal since it only succeeds if an exact center is found, instead of looking for the cell closest to the center. How would you fix it?

```
      miny := maxy := nextpiece[1][1]
      minx := maxx := nextpiece[1][2]
      every miny >:= (!nextpiece)[1]
      every minx >:= (!nextpiece)[2]
      every maxy <:= (!nextpiece)[1]
      every maxx <:= (!nextpiece)[2]
      every i := 2 to *nextpiece do
        if nextpiece[i][1] == (miny + maxy) / 2 &
           nextpiece[i][2] == (minx + maxx) / 2 then
          nextpiece[1] :=: nextpiece[i] # swap!
      if miny < 1 then every (!nextpiece)[1] +:= −miny + 1
      every minx to 3 do every (!nextpiece)[2] +:= 1
      if (!nextpiece)[1] > 5 then return newobject()
      if (!nextpiece)[2] > 8 then return newobject()
      }
```

In contrast to the random shapes, the standard shapes use hardwired coordinates and colors.

```
   else case nextcolor := ?["red−yellow", "red", "yellow",
"green",
                            "cyan", "blue", "purple−magenta"] of {
   "red−yellow":     nextpiece := [ [1,5], [1,6], [2,5], [2,6] ]
   "yellow":         nextpiece := [ [2,6], [1,6], [2,5], [2,7] ]
   "blue":           nextpiece := [ [2,6], [1,5], [2,5], [2,7] ]
   "purple−magenta": nextpiece := [ [2,6], [1,7], [2,5], [2,7] ]
   "red":            nextpiece := [ [3,6], [1,6], [2,6], [4,6] ]
   "green":          nextpiece := [ [2,6], [1,5], [1,6], [2,7] ]
   "cyan":           nextpiece := [ [2,6], [1,6], [1,7], [2,5] ]
   }
end
```

Procedure `move_piece(x,y)` moves the active cells for the current object by an offset (x,y) that handles movement left and right, as well as down. The new desired location of all the cells is calculated, and then procedure `place_piece()` is called to try to put the piece at the new location.

```
procedure move_piece(x, y)
   newactivecells := []
   every cell := !activecells do
      put(newactivecells, [cell[1]+y, cell[2]+x])
   return place_piece(newactivecells, x)
end
```

The `place_piece()` procedure checks whether the object can go into the location proposed, and if it can, it draws the piece in the new location and updates the `activecells` variable. Testing whether parameter `horiz = 0` allows the code to skip redrawing the object's "footprint" below the play area in the common case when the object just drops one row. The backslash in `\horiz` causes the expression to fail if `horiz` is null, so the second parameter may be omitted.

```
procedure place_piece(newactivecells, horiz)
   if collision(newactivecells) then fail
   if not (\horiz = 0) then
      EraseArea(120,481,150,15)
   every cell := !activecells do {
      FillRectangle(colors["black"],
                  120 + (cell[2]-1)*15, 20 + (cell[1]-1)*15,15,15)
      L[cell[1], cell[2]] := "black"
      }
   every cell := !newactivecells do {
      L[cell[1], cell[2]] := activecellcolor
      drawcell(120 + (cell[2]-1)*15, 20 + (cell[1]-1)*15,
               activecellcolor)
      if not (\horiz = 0) then
         drawcell(120 + (cell[2]-1)*15, 481, activecellcolor)
      }
   WSync()
   activecells := newactivecells
   return
end
```

Procedure `collision()` checks each cell that the object is moving into to see if it is occupied by something other than part of the currently active piece. Check out the cool "break next" expression when a black cell is found. It exits the inner loop and skips to the next iteration of the outer loop, preventing some "false positive" collision results..

```
procedure collision(cells)
   every c := !cells do {
      if not ((1 <= c[1] <= 30) & (1 <= c[2] <= 10)) then return
      if L[c[1], c[2]] === "black" then next
      every a := !activecells do {
         if (c[1] = a[1]) & (c[2] = a[2]) then
            break next
         }
      if L[c[1], c[2]] ~=== "black" then return
      }
   fail
end
```

Rotating a piece is similar to moving it, in that it involves calculating a new location for each cell. The first active cell in the list is considered the "center" around which the rest of the cells rotate. To rotate a cell by ninety degrees around the center, compute its x− and y−coordinate offsets from the center cell. The cell's rotated position uses these same

offsets, but swaps the x−offset with the y−offset, and reverses the signs of one offset or the other, depending on which quadrant the rotation is coming from and going into. Figure 16−2 shows how the signs are reversed depending on the quadrant.

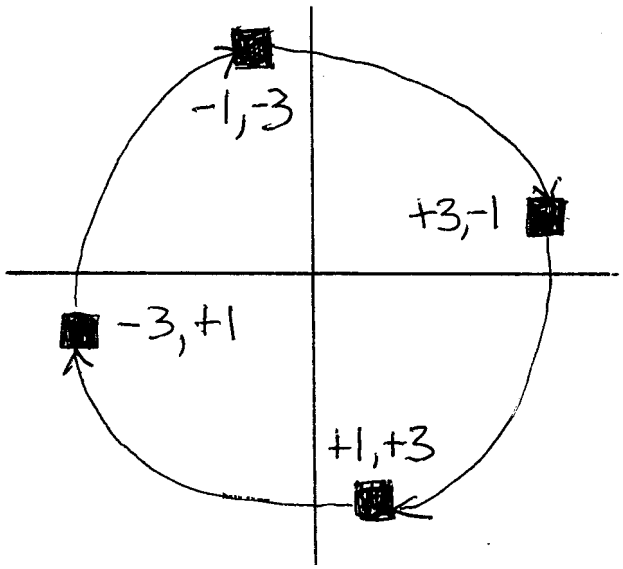

Figure 16−2:
Rotating a cell swaps its x and y offsets, with sign changes

Four standard cell shapes, identified by color, are handled specially during rotation. Squares (red−yellow) have no rotation. The other three standard cell types (red, green, and cyan) are symmetric shapes whose rotation appears smoother if it alternates clockwise and counterclockwise. This alternation is handled by global variable `pieceposition`.

```
procedure rotate_piece(mult1, mult2)
   if activecellcolor === "red-yellow" then fail
   newactivecells := list()
   centerpoint := copy(activecells[1])
   differencelist := list()
   every point := ! activecells do {
     temp := [centerpoint[1] - point[1], centerpoint[2] -
point[2]]
      put(differencelist, temp)
      next
      }
   every cell := !activecells do
      put(newactivecells, copy(cell))
   if activecellcolor === ("red" | "green" | "cyan") then {
      if pieceposition = 2 then {
         mult2 :=: mult1
         pieceposition := 1
         }
      else pieceposition := 2
      }
   every foo := 1 to *newactivecells do
      newactivecells[foo] := [
         centerpoint[1] + differencelist[foo,2] * mult1,
         centerpoint[2] + differencelist[foo,1] * mult2
         ]
   return place_piece(newactivecells)
end
```

Each time a piece stops falling, procedure `scanrows()` checks to see whether any rows in the playing area are filled and can be removed. If no black is found on a row, that row is put on a list called `rows_to_delete`.

The player scores $50 * 2^{k-1}$ points for k deleted rows. To maximize your score you should try to always delete several rows at once!

```
procedure scanrows()
   scanned_rows := table()
   rows_to_delete := []
   every point := !activecells do {
      if \scanned_rows[point[1]] then next
      scanned_rows[point[1]] := 1
      every x := 1 to 10 do {
         if L[point[1], x] === "black" then
            break next
         }
      put(rows_to_delete, point[1])
      }
   if *rows_to_delete > 0 then {
      Fg("black")
      drawstat(score, numrows, level, tot_score)
      numrows +:= *rows_to_delete
      level := integer(numrows / 10)
      score +:= 50 * (2 ^ (*rows_to_delete - 1))
      tot_score +:= 50 * (2 ^ (*rows_to_delete - 1))
      delaytime := 200 - (10 * level)
      Fg("white")
      drawstat(score, numrows, level, tot_score)
      deleterows(rows_to_delete)
      }
end
```

The code to delete rows takes the list of rows to delete, sorts it, and builds a corresponding set. It then moves the bottom rows of L, up to the first row to be deleted, into a temporary list. For each row in the temporary list, if it is to be deleted, a new row of black cells is inserted at the top of L, otherwise the row is reappended to the end of L. When the play area has been reassembled it is redrawn.

```
procedure deleterows(rows_to_delete)
   temp := []
   rows_to_delete := sort(rows_to_delete)
   row_set := set()
   every insert(row_set, !rows_to_delete)
   current_row := 30
   while current_row >= rows_to_delete[1] do {
      push(temp, pull(L))
      current_row -:= 1
      }
   current_row := 1
   basesize := *L
   while *temp>0 do {
      if member(row_set, basesize + current_row) then {
         push(L, list(10, "black"))
         pop(temp)
         }
      else
         put(L, pop(temp))
      current_row +:= 1
      }
   refresh_screen()
   WSync()
end
```

Sesrit provides several buttons to the left of the play area that allow the user to pause, quit, start a new game, or see author information. The procedure `buttons()` is called to handle input events whenever the game is not actually running, such as before it starts or when it is paused. Its code is analogous to the user input handling in `game_loop()` and is not shown here. The procedure `about_sesrit()` implements the about box, a simple dialog that shows author information until the user dismisses it. It illustrates several graphics library procedures related to drawing of text. `CenterString()` is a useful library procedure that draws a string centered about an (x,y) location.

```
procedure about_sesrit()
   about := WOpen("label=About Sesrit", "size=330,200",
"fg=white",
                  "bg=black", "posx=10", "posy=155") | fail
   Bg("black")
   every cell := !nextpiece do
      EraseArea(-40 + (cell[2]-1)*15, 60 + (cell[1]-1)*15, 15,
15)
   FillRectangle(colors["black"], 120,20,150,450, 120,481,150,15)
   CenterString(about, 165, 25, "Written By: David Rice")
   CenterString(about, 165, 50,
                "Communications Arts HS, San Antonio")
   CenterString(about, 165, 90, "and")
   CenterString(about, 165, 115, "Clinton Jeffery")
   CenterString(about, 165, 180, "Spring 1999")
   Event(about)
   while get(Pending())
   WClose(about)
   if game_status = 1 then refresh_screen()
end
```

The last procedure from *Sesrit* that we will present is the one that redraws the play area, `refresh_screen()`. It draws the next piece cells to the left of the play area, it draws the footprint outline of the active cell below the play area, and then it draws the entire play area with a big loop that draws filled rectangles. Cell `L[x,y]`'s colors are looked up in the color table to determine the color of each rectangle that is drawn.

```
procedure refresh_screen()
   every cell := !nextpiece do
      drawcell(-40 + (cell[2]-1)*15, 60 + (cell[1]-1)*15,
               nextcolor)
   every cell := !activecells do
      drawcell(120 + (cell[2]-1)*15, 481, activecellcolor)
   every (x := 1 to 30, y := 1 to 10) do
      drawcell(120 + (y-1)*15, 20 + (x-1)*15, L[x, y])
end
```

Galactic Network Upgrade War: a Strategy Game

The next example is an illustration of object−oriented design for a complex strategy game of a depth competitive with commercial offerings. The scope of this example is enormous, and this chapter only presents highlights from the design of this 10,000+ line program. The focus is on the challenge of implementing an object−oriented design in Icon. Like many large programs, this game is a "living thing" that is never complete and is slowly evolving.

This game is a parody of almost everything, from the software industry to the political system to the great science fiction epics of the twentieth century. It owes some direct inspirational credit to a charming old war game called *Freedom in the Galaxy*, designed by Howard Barasch and John Butterfield, and published originally by Simulations

Publications, Inc. and later by the Avalon Hill Game Company. Freedom in the Galaxy is in turn a homage (or parody) to George Lucas' *Star Wars* films. Our game is a bit farther over−the−top than either of these fine works.

The Play's the Thing

What better medium for setting the mood, than epic poetry?

> ***The Lord of the Nets***
>
> *Green nets to snare compiler wizards from ivory towers in deep space,*
> *Gold nets to lure database gurus, their galactic servers in place,*
> *Silver nets for IT media pundits and their damned lies,*
> *A black net for the Dark Lord, to mask his dark face.*
>
> *One Net to take them all,*
> *One Net to buy them,*
> *One Net to tax them with*
> *And in the license bind them*
> *In the land of Redmond, where Executives lie.*

Galactic Network Upgrade War (GW) is an epic space−fantasy strategy game in which an evil empire holds an entire galaxy in its iron grip by means of monopolistic software practices. A small band of rag−tag independent software developers fight a hopeless crusade against the ruthless tyrant, the Microsaur Corporation and its Dark Lord, Microsauron.

You are a young hacker who must work your way up the ranks, on the side of good or evil, until you earn the respect and influence that enable you to lead your side to victory, crush your opponents, and gain control over the galactic internet.

Background

In the year 9,999, as the entire galaxy struggles for its very survival in the throes of the Y10K bug, one company holds all the cards. Governments have long since defaulted on their loans and been replaced by the rule of the multi−stellar corporations. After a time of cataclysmic battles, all of the surviving corporations swore fealty to one ruler, becoming subsidiaries with varying nano−degrees of freedom. That ruler was the Microsaur Corporation.

Some say Microsaur took over through superior innovation, blending the best aspects of biological and technological species into a super−race, a hybrid of voracious reptile and implanted artificial intelligence. Others say that the Microsaur Collective, a predator unchecked, simply ate all organized resisters on the galactic network. Perhaps both accounts are true. In any case, Microsaur Corporation took over by controlling the amalgamated IT resources of the entire galaxy, on every desk in every home and office.

But although organized corporate resistance was futile, lonely bands of mysterious rebels persisted in conveying an oral tradition of a time when machines ran open source software. Unable to maintain any presence on digital media, these groups, such as the Gnights who say Gnu, memorized entire programs such as Gnu Emacs in their heads and passed them on to their children for generations. This went on until one day Tux the

Killer Penguin inexplicably appeared in a puff of logic on an unguarded workstation at the edge of the galaxy in the office of one Doctor Van Helsingfors. The Microsaur collective's presence on that machine flickered out too instantaneously and too faintly to even be noticed in the corporate campus at the center of the galactic Internet. Finding him playful, cuddly, and more amusing than the Microsaur that Tux had slain, Doctor Van Helsingfors shared Tux with a few thousand of his closest friends, little realizing his communiqué would change the galaxy forever.

Tux the Killer Penguin was friendly to Sapes and other bio life forms, but had a rare property: an insatiable hunger for Microsaurs. Tux was a network aware, open system artificial intelligence (an "Aint," as they are called). With the assistance of Sape (from Homo sapien) rebels, Tux spread like a virus through the galactic network, paving the way for other open system software on small machines across the galaxy. Eventually, the Microsaur collective detected that it no longer held its 100 percent market share, and once it identified Tux, it began the most radical binary purge in galactic history, expecting little or no resistance. Thus begins the epic war that you are about to participate in.

Software Architecture

GW could benefit from a client−server architecture of the sort described in Chapters 5 and 14, but because it is already complex in so many other areas, it utilizes a single process model in which players' screens are opened on different displays at program startup. This relies on client−server capabilities that are an implicit part of Icon's graphics facilities, delivered by the X Window System; although, GW will run on a Windows machine, its multiplayer capabilities do not function there.

GW has an essentially sequential, turn−based model of time. During each turn, there is an orders phase followed by a resolution phase. In the orders phase, players specify movements, combats, and/or special activities for the individuals and armies under their control. When all players are finished giving orders, the resolution phase shows them the outcome of the players' efforts: their own and their opponents'.

Use Cases for Game Activities

The design of a complex piece of software such as GW starts with an identification of the use cases, as described in Chapter 11. What actors does the game have, and what tasks do they perform? Since GW is a game, it is obvious that the tasks associated with playing the game had better be fun, or there is little point in creating the software, except as an illustration of object oriented techniques!

There is only one type of actor in GW: a Player. Players interact with the software to perform many different tasks. Figure 16−3 shows some of these tasks. If you get picky, there are two kinds of Players, those who work for Microsauron, and those who don't, and there are some tasks that are only performed by one side or the other, but this chapter will ignore those distinctions.

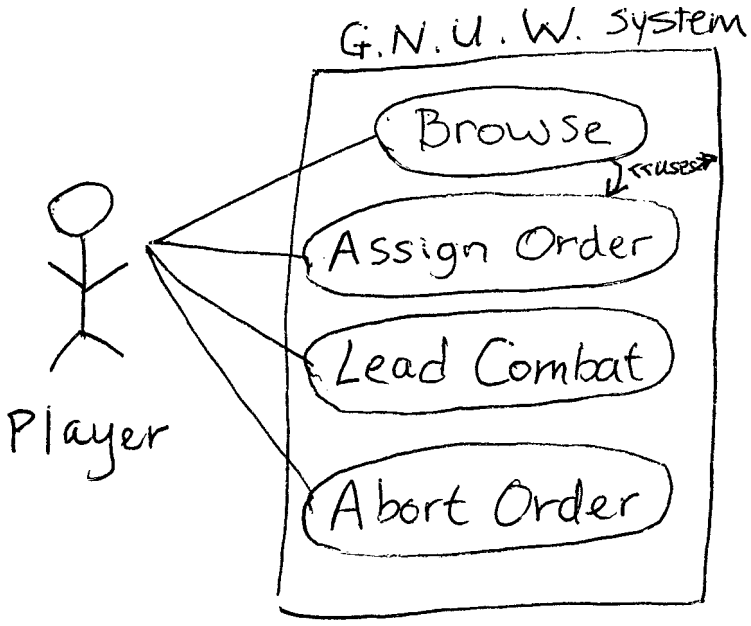

Figure 16−3:
Use Case Diagram for Game Activities

During the orders phase, the primary task might be termed **Browse**, since the player is examining the current orders (carried over from the previous turn by default) and deciding what to change. The browsing task uses, repeatedly, a subtask **Assign Order**. This subtask may be as simple as clicking on the map to select a group, and pressing a key shortcut that assigns a particular order to that group. Order assignment may be more complex, however, involving a dialog that specifies details about an order's properties.

During the resolution phase, the player is primarily receiving information from the computer about the outcome of his or her efforts. However, there are several tasks that are required during this process. If a player's characters have a hostile encounter, the player must **Lead Combat**. A player may elect to **Abort Order** when they determine their characters are in danger. Similar tasks are associated with military forces.

These cleverly named use cases are meaningless without a textual description of the sequence of actions a player performs to complete each task. The use case descriptions are given in a consistent format starting with a general description, initiating and terminating conditions, and then the sequence of steps, including branching alternatives, performed by the player when completing that task.

Browse

Initiated: at the beginning of each game turn

Completed: when all users have pressed the Make It So button.

Browsing consists of navigating the galaxy, assigning orders to ones' units (characters and military forces). The sequence of steps consists of

1. Navigate among views until a location of interest is selected.
2. Inspect and possibly modify unit grouping in that location.

3. Inspect and possibly modify each group’s orders.
4. If satisfied, press "Make It So"; otherwise, go to step 1.

Assign Order

Initiated: by the user right clicking on a selected group while browsing

Completed: when the user clicks OK or Cancel on the Assign Orders dialog.

Assigning orders is usually a matter of selecting from a list. The sequence of steps consists of

1. Display current/previous orders details for a selected group.

2. Select from among the following orders (this list is representative):

> **Move** to location X. Digital units can move instantly to any location connected to the Galactic Internet. Biological units may take several turns to cross a desired space, depending on the craft in which they travel.
>
> **Search** for (group, location, or artifact)
>
> **Attack** group X with force or by virus (biological or digital)
>
> **Sabotage** a planet’s defenses or a solar system’s internet gateway
>
> **Evangelize** an enemy or the community in a location to extol your side or expose the technically inferior or evil nature of the opposing side.

3. If satisfied with orders as specified, press OK or Cancel, otherwise go to step 2.

Lead Combat

Initiated: as needed during the resolution phase of each game turn, in locations occupied by both sides, when one or the other side has ordered an attack.

Completed: when the combat is over.

Leading combat consists of supplying tactical directions through one or more rounds of combat. There are three kinds of combat: military, melee, and digital. The sequence of steps consists of

1. Select whether to attack, defend, or flee.
2. Observe the results of the round.
3. If both sides are still present in the combat, go to step 1.

Abort Order

Initiated: during the resolution phase, when things are going badly.

Completed: instantaneously.

If a group has sustained too many casualties to complete their orders, the player may attempt to abort. For example, if a group trying to evangelize a planet is attacked and injured, they may attempt to skip any further risk associated with evangelism this turn.

An Object Model for a Galaxy

GW has a complex model of a galaxy that represents enough information to allow objects representing military forces and individual heroes to move about, fight, and go on adventures. Unlike many games, whose "spaces" in which pieces move are based on a grid of cells or a circular track, the Galaxy in GW is represented by a graph of nodes, each one representing a solar system capable of supporting higher life forms. The interconnections between nodes in the graph represent units of distance that may be traversed by faster−than−light travel. Each node contains some number of habitable planets, consisting of one more habitats in which the game's pieces may be located during any particular turn.

Groups of neighboring solar systems together form quadrants, and the galaxy is just a collection of quadrants. The entire galaxy, from quadrants and solar systems down to the details of planets and habitats is generated at random the first time the program runs. It may be regenerated from the main menu. Figure 16−4 shows the object model for the game's model of the universe. The aggregation diamonds are drawn solid, indicating an immutable part−whole relationship.

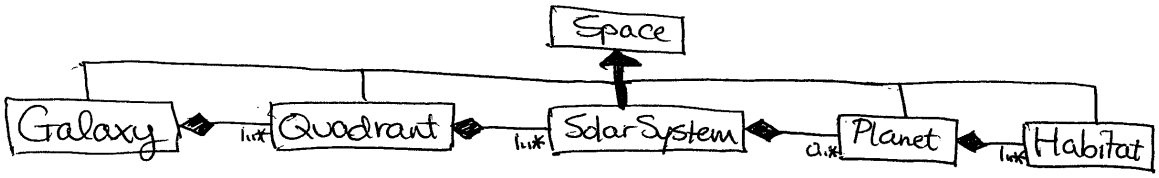

Figure 16−4:
Object Model for a Galaxy

Within this universe of places, the object model in Figure 16−5 shows relationships between the persons and groups that are manipulated during play. Characters and military forces comprise discrete entities called units. Units are routinely grouped together for common purposes; a group is a set of units given common orders. The aggregations in this diagram are drawn hollow to reflect dynamic, changing relationships. The location of each group within a habitat or other kind of space is marked as a user−defined association, as is the ability of characters to lead military groups.

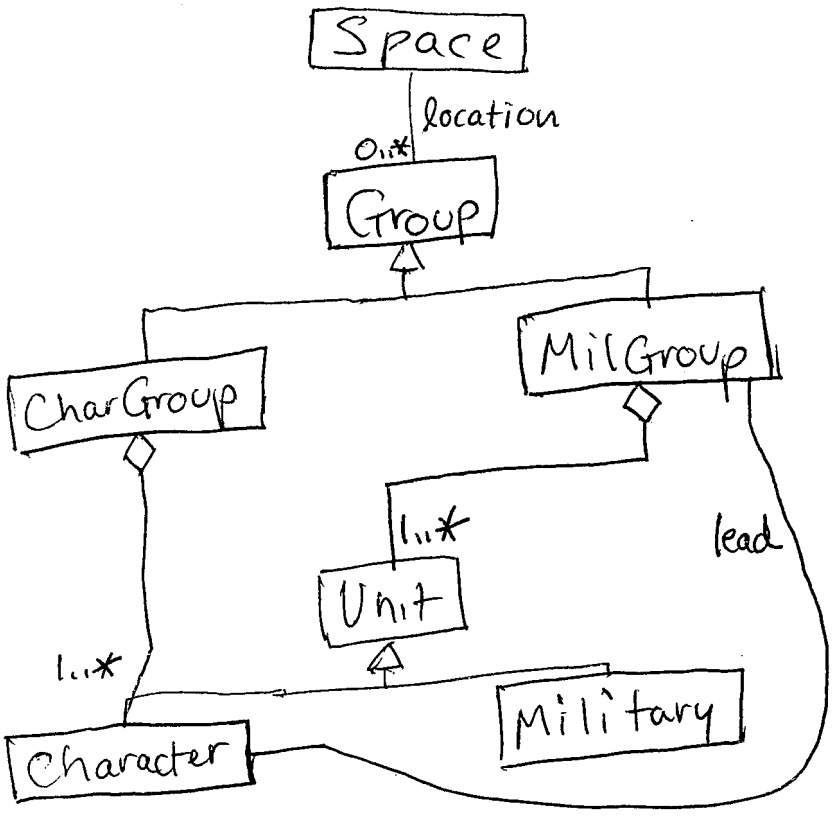

Figure 16−5:
Groups of Units Perform Game Tasks in Various Locations

The Map

Each player gets a different view of the galaxy, reflecting both his or her own focus of attention and the limited information available to them. Their map is drawn on a window, and contains glyphs (visual objects, or symbols) corresponding to a subset of the objects in the galaxy. In general, several players may have different glyph objects (with different location, size, or other attributes) that refer to the same underlying galaxy object. Figure 16−6 shows the organization of glyphs on a player's map.

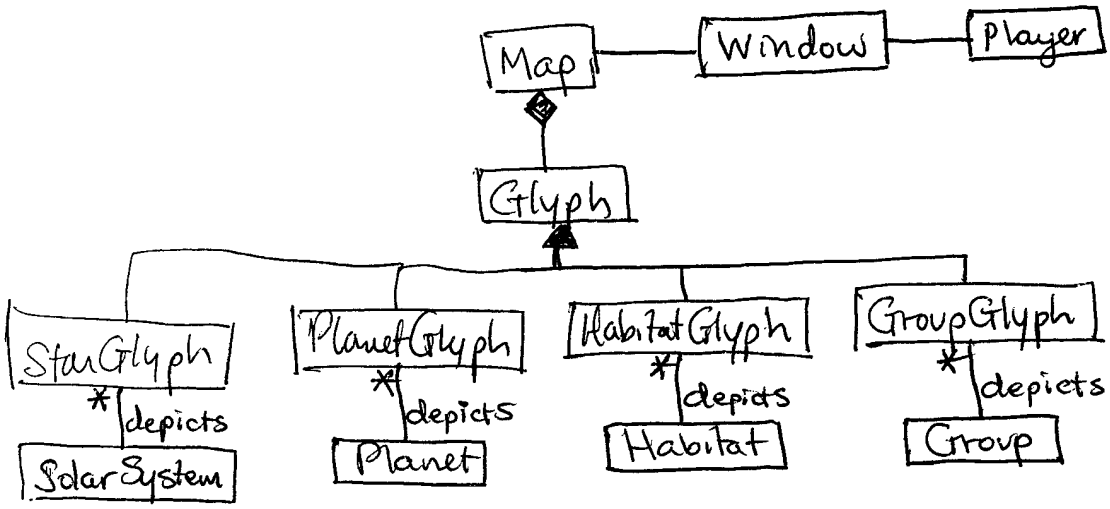

Figure 16−6:
The Map's Glyph Objects Depict Spaces and Groups

The User Interface

GW's user interface centers around a map of the galaxy that depicts everything the player knows about the movements and activities of friendly and hostile forces. With limited screen space, it is tempting to show only one piece of information at a time to present it in detail, say for example, one habitat (or other type of space object, with its contents) at a time. The problem with this is that strategic decisions about a group's activities requires more information than that provided in a single habitat; strategy requires a global view.

Another way to state the problem with this type of "modal" interface is that the player will be constantly switching modes and reorienting themselves when the view changes. Instead of this, GW's map uses a technique called a *graphical fisheye view* to show multiple focus star systems in detail, while preserving a global view of the system. Figure 16−7 shows a view of a galaxy. A light (yellow) circle, orbited by one or more planets depicts each solar system. Solar systems of interest are drawn in vastly more detail than others, controlled by clicking the left (enlarge) and right (shrink) mouse buttons on the different stars. The names of stars or planets are drawn in varying font sizes, as long as sufficient space is available. Planet details such as habitats, defenses, and loyalties are depicted. The simple colors and textures used for stars and planets are legible and easily learned, but these prototype−quality graphics should be upgraded with more artistic or photo−like images of astral bodies.

Figure 16−7:
The Graphical Fisheye View in GW

GW’s other activities, such as giving orders, can generally be performed using keyboard shortcuts, or using various dialogs that present detail information. Figure 16−8 shows a dialog used for assigning orders within a habitat. The units are depicted with humorous mock−up images and various attribute details in the upper area of the window. The bottom center contains a list of orders these characters are eligible for in this habitat. The currently selected order is described in detail in the lower right area of the dialog. As play commences, players gain or lose characters and military units, and separately, they gain or lose installed base and "mind share" for their software’s cause on each planet in the galaxy. The player can define winning in terms of survival, domination, or eradication of all opposition.

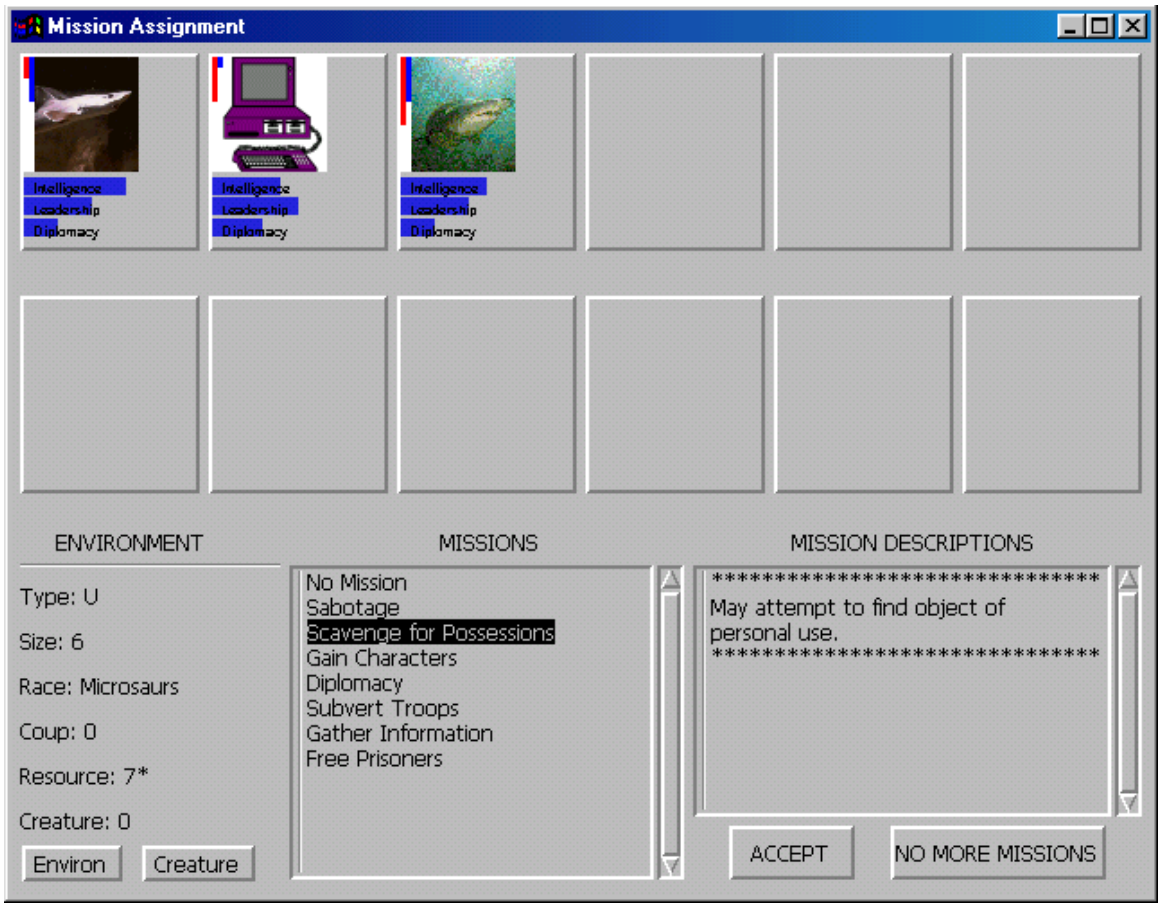

Figure 16−8:
Assigning Orders Dialog

Summary

Icon is a fun language to use for writing many different kinds of games. Games usually make use of graphics facilities and increasingly, the Internet facilities available in Unicon as well. Thanks to the high score server from Chapter 14, any game that includes a point−scoring system can easily provide global competition for players with Internet access.

For complex games, designing the game may be as hard a job as programming it. It helps to start from a complete prose description of all game activities, to develop use cases, and then to describe the kinds of objects and relationships between objects that will be necessary for those game activities to occur. This chapter is able to suggest that process, but to show it in detail would be the subject of another entire book.

Chapter 17: Object−oriented User Interfaces

Most application programs interact with users through a graphical user interface. To achieve this, programmers use toolkits such as Motif or class libraries such as Swing or MFC. While Icon's platform−independent graphics facilities are excellent for drawing to the screen, manipulating windows, and so on, the standard elements of graphical user interfaces are not built−in to the language. A library of procedures called the vidgets library provides Motif−like buttons, menus, scroll bars, and so on. The vidgets library suffers from two drawbacks: it is missing many interface components that are used in modern applications, and it is not easily extended to include new components.

This chapter presents an alternative user interface class library that addresses these drawbacks. An object−oriented design is used to reduce complexity and increase the extensibility of the toolkit. The result is an elegant library, simply called "the GUI toolkit," that comes in a single source file, gui.icn. The GUI toolkit is a class library, and as such it is specific to Unicon, and not intended to address the needs of Arizona Icon programmers, who can use the vidgets library. Despite being only around two−thirds of the size (in lines) of the vidgets library, the GUI toolkit offers many more interface components. Although this book describes the components provided by the GUI toolkit, you will need the book *Graphics Programming in Icon* [Griswold98] to develop custom graphical user interfaces.

To examine the capabilities of the GUI classes, the chapter presents an example program that allows you to design an interface by visually creating a dialog on the screen interactively. The program then generates an Unicon program that can be filled in to create a working application. When you finish this chapter you will know 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 ivib, the Unicon visual interface builder.

A Simple Dialog Example

Object−orientation seems to be a big help in designing graphical user interfaces, and this is as true for Icon as it is for other languages. The best way to see how the GUI classes work is to try out a simple example program. Listing 17−1 shows the source code in full, which is described in detail below.

Listing 17−1
The TestDialog Program

```
link gui
class TestDialog : _Dialog(b)
   method dialog_event(ev)
      if ev.get_component() === b & ev.get_code() > 0 then
         dispose()
   end
initially
   local l
```

```
   self._Dialog.initially()
   l := Label("label=Click to close", "pos=50%,33%", "align=c,c")
   add(l)
   b := TextButton("label=Button", "pos=50%,66%", "align=c,c")
   add(b)
   set_attribs("size=215,150", "bg=light gray",
               "font=serif", "resize=on")
   show()
end

#
# Program entry point.
#
procedure main()
   TestDialog()
end
```

Assuming the program is stored in a file called `eg1.icn`, issue the following command to compile it:

```
unicon eg1
```

The result should be an executable file called `eg1`. Run this program, and the window shown in Figure 17−1 should appear, and should close when the button is clicked.

Figure 17−1:
TestDialog window

It is worth examining this example program line by line. It begins by declaring a new class, `TestDialog`. The line

```
class TestDialog : _Dialog(b)
```

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 variable `b` represents the button in the dialog box, and is given a value later.

Next comes a definition of a method called `dialog_event()`. Every dialog must define this method, which is invoked by the toolkit whenever an event of interest has occurred. You can consider the contents of the method in a moment, but first look at the remainder of the `TestDialog` class, which is contained in its `initially` method. As you will recall, the `initially` method is called once only, when a class instance is created. The `initially` method begins with the line

```
self._Dialog.initially()
```

This, like the `dialog_event()` method, is obligatory and must appear for each dialog. Next comes code that adds the label to the dialog:

```
l := Label("label=Click to close", "pos=50%,33%", "align=c,c")
add(l)
```

This assigns the variable `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 specified as 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 sets the attributes of the dialog window, including its initial size. Try changing these values to experiment with other dialog styles.

```
set_attribs("size=215,150", "bg=light gray",
            "font=serif", "resize=on")
```

The last line is

```
show()
```

This actually displays the dialog window and goes into the toolkit's event handling loop, from which `dialog_event()` is called when an event occurs. Now, go back to look at the `dialog_event()` method of the test program:

```
method dialog_event(ev)
   if ev.get_component() === b & ev.get_code() > 0 then
      dispose()
end
```

The variable `ev` is an instance of a class called `_Event`. The `dialog_event()` method uses two methods of that class. The first, `get_component()`, returns the component which has generated this particular event. Therefore we test to see if it is our button `b`. The second method, `get_code()`, returns an integer value. This tells us which one of several events has occurred to the component. A button generates the following codes for example:

Code	Event
0	Button pressed down
1	Button released
2	Return key pressed while button has focus

Positioning Objects

The button and the label were positioned in our above example by specifying 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 class constructor strings, with x and y values separated by commas. By default, the position specified relates to the top left corner of the object concerned. You can use the `"align"` attribute, which takes two alignment specifiers, to change this default. The first alignment specifier is an `"l"`, `"c"`, or `"r"`, for left, center, or right horizontal alignment, respectively; the second is a `"t"`, `"c"`, or `"b"`, for top, center, or bottom vertical alignment, respectively. There is one further attribute, `"size"`, whose specifiers 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 will default to a size based on the font in use and the string in the button.

A generic method `attrib(attribs[])` implements attribute processing, but it mostly farms out the 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)`. These methods may be used to effect dynamic changes, as opposed to static interfaces whose values are fixed at the time of object−creation.

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, and takes up 80 percent of the width; vertically its top edge 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.

A More Complex Dialog Example

Now it's time to introduce some more component types. Listing 17−2 shows our next example program in full.

Listing 17−2
SecondTest Program

```
#
# Second test program
#
class SecondTest : _Dialog(
   #
   # The class variables; each represents an object
   # in the dialog.
   #
   text_menu_item_1, text_menu_item_2,
   check_box_1, check_box_2, check_box_3,
   text_list,
   tabl, table_column_1, table_column_2,
   lst,
   text_field,
   check_box_group,
   label_2, label_1, label_5, label_3, label_4,
   #
```

```
# Some data variables.
#
oses,
languages,
shares
)

#
# Add a line to the end of the text list
#
method put_line(s)
   local l
   l := text_list.get_contents()
   put(l, s)
   text_list.set_contents(l, *l)
end

#
# Default event handler
#
method handle_default(ev)
   local icon_event
   icon_event := ev.get_event()
   put_line("Icon event " || icon_event)
   #
   # Check for Alt-q keypress
   #
   if icon_event === "q" & &meta then
      dispose()
end

#
# The quit menu item
#
method handle_text_menu_item_1(ev)
   dispose()
end

#
# Other event handlers - print a line of interest to
# the text list.
#
method handle_check_box_1(ev)
   put_line("Favorite o/s is " || oses[1])
end

method handle_check_box_2(ev)
   put_line("Favorite o/s is " || oses[2])
end

method handle_check_box_3(ev)
   put_line("Favorite o/s is " || oses[3])
end

method handle_text_field(ev)
   put_line("TextField code = " || ev.get_code())
   put_line("Contents = " || text_field.get_contents())
end

method handle_list(ev)
   put_line("Favorite language is " ||
            languages[lst.get_selection()])
end

method handle_text_menu_item_2(ev)
```

```
      put_line("You selected the menu item")
   end

   method handle_table(ev)
      local i
      i := tabl.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

   #
   # Event handler - just pass on events to other methods.
   #
   method dialog_event(ev)
      case ev.get_component() of {
         text_menu_item_1 : handle_text_menu_item_1(ev)
         text_menu_item_2 : handle_text_menu_item_2(ev)
         check_box_1 : handle_check_box_1(ev)
         check_box_2 : handle_check_box_2(ev)
         check_box_3 : handle_check_box_3(ev)
         tabl : handle_table(ev)
         table_column_1 : handle_table_column_1(ev)
         table_column_2 : handle_table_column_2(ev)
         lst : handle_list(ev)
         text_field : handle_text_field(ev)
         default : handle_default(ev)
         }
   end

   initially
      local menu_bar, menu,
            panel_1, panel_2, panel_3, panel_4, panel_5
      self._Dialog.initially()

      #
      # 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 attribs and minimum size.
      #
      set_attribs("size=568,512", "font=sans",
                  "bg=light gray", "resize=on")
      set_min_size(494, 404)

      #
      # Set up a simple menu system
      #
      menu_bar := MenuBar()
      menu := Menu("label=File")
      text_menu_item_1 := TextMenuItem("label=Quit")
      menu.add(text_menu_item_1)
      text_menu_item_2 := TextMenuItem("label=Message")
```

```
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=5%,10%", "size=134,132")
label_2 := Label("pos=10,10", "ialign=l",
                 "label=Favorite o/s")
panel_1.add(label_2)
check_box_1 := CheckBox("pos=10,40", "label=" || oses[1])
check_box_group.add(check_box_1)
panel_1.add(check_box_1)
check_box_2 := CheckBox("pos=10,70", "label=" || oses[2])
check_box_group.add(check_box_2)
panel_1.add(check_box_2)
check_box_3 := CheckBox("pos=10,100", "label=" || oses[3])
check_box_group.add(check_box_3)
panel_1.add(check_box_3)
add(panel_1)

#
# The text-list of messages.
#
panel_2 := Panel("pos=45%,10%", "size=260,192")
text_list := TextList("pos=10,40", "size=240,135")
text_list.set_contents([])
panel_2.add(text_list)
label_1 := Label("pos=10,10", "ialign=l", "label=Messages")
panel_2.add(label_1)
add(panel_2)

#
# The table of shares.
#
panel_3 := Panel("pos=45%,55%", "size=260,192")
tabl := Table("pos=10,40", "size("240", "127")
tabl.set_select_one()
tabl.set_contents(shares)
table_column_1 := TableColumn("label=Company",
                              "ialign=l", "colwidth=100")
tabl.add(table_column_1)
table_column_2 := TableColumn("label=Share price",
                              "ialign=r", "colwidth=100")
tabl.add(table_column_2)
panel_3.add(table)
label_5 := Label("pos=10,10", "ialign=l", "label=Shares")
panel_3.add(label_5)
add(panel_3)

#
# The drop-down list of languages.
#
panel_4 := Panel("pos=5%,45%", "size=182,83")
lst := List("pos=10,40", "size=100")
lst.set_selection_list(languages)
panel_4.add(list)
label_3 := Label("pos=10,10", "ialign=l",
                 "label=Favorite language")
panel_4.add(label_3)
add(panel_4)

#
```

```
        # The text field.
        #
        panel_5 := Panel("pos=5%,70%", "size=182,83")
        label_4 := Label("pos=10,10", "ialign=l",
                         "label=Enter a string")
        panel_5.add(label_4)
        text_field := TextField("pos=10,40", "size=129,")
        text_field.set_draw_border()
        text_field.set_contents("")
        panel_5.add(text_field)
        add(panel_5)

        #
        # Display the dialog
        #
        show()
end

#
# Simple main procedure just creates the dialog.
#
procedure main()
    SecondTest()
end
```

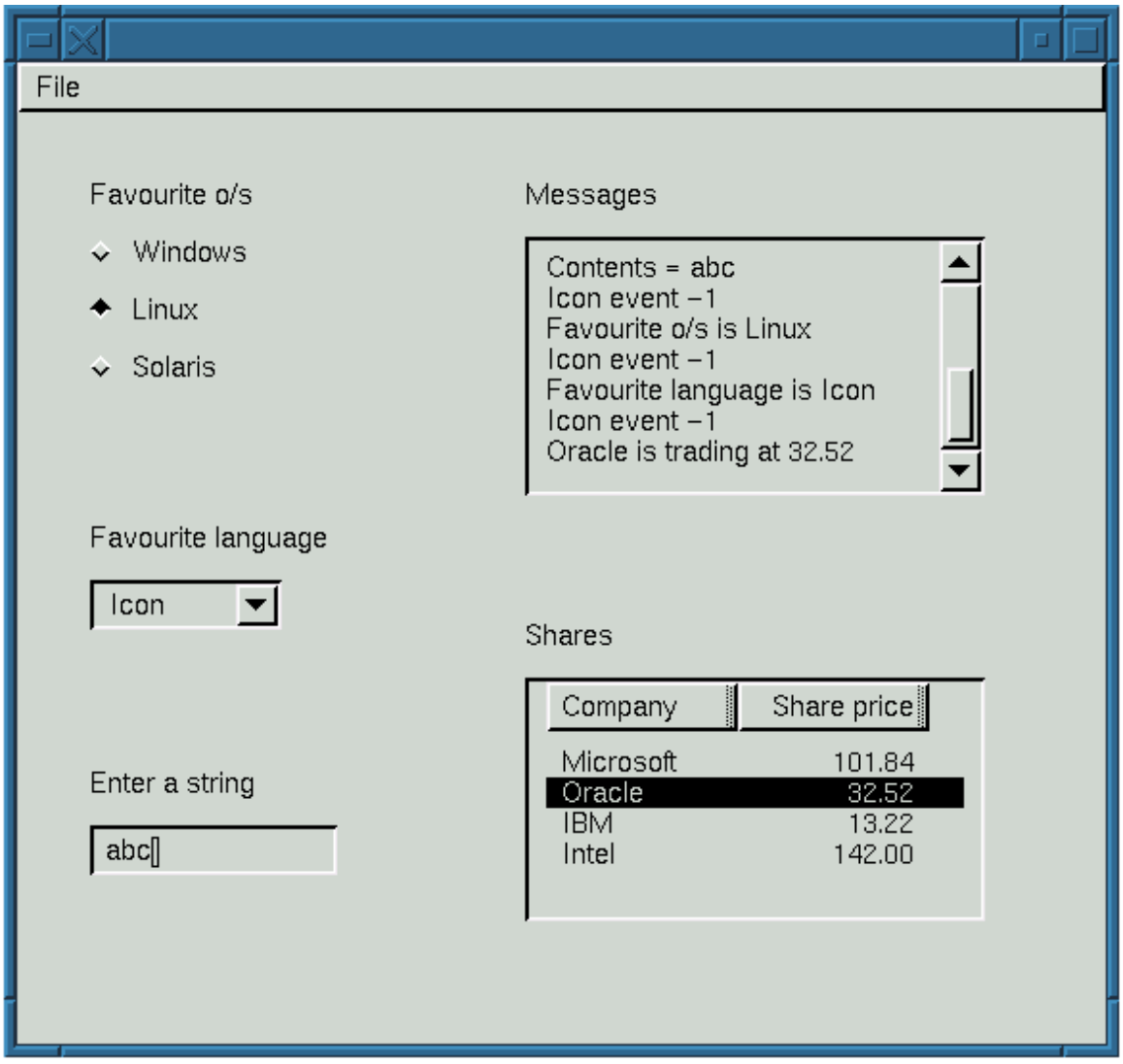

Figure 17−2:
SecondTest window

Start your look at this program with the `initially` method at the end. This method begins by initializing some data and setting the attributes. Then, the minimum size of the window is set with

```
set_min_size(494, 404)
```

The window cannot be made smaller than these dimensions.

The next part creates a menu bar structure. We will deal with how menu structures work in detail later in this chapter, but for now just 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.

The next section sets up the three check boxes (also known as radio buttons). These are placed, together with the label "Favorite o/s" in a `Panel` object. This object simply serves as a container for other objects that logically can be treated as a whole. The most important thing to note about a `Panel` is that objects within it 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=10, 10"`. This means ten pixels relative to the top left−hand corner of the `Panel`, not ten pixels relative to 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 coordinate themselves as a group so that when one is checked another is unchecked, they are placed together in a `CheckBoxGroup` object. This does no more than "bracket" them together. Note that each `CheckBox` is added to the `CheckBoxGroup` and the `Panel`.

The next section is another `Panel` that holds another label ("Messages") and a `TextList` object. In this case the object is used to hold a list of message strings that scroll by like a terminal window. A `TextList` object can also be used for editing text, or for selecting one or more items from a list of strings.

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. The table columns have their initial column width specified with attribute `colwidth` and the alignment of the column's contents set with attribute `ialign`. The method `set_select_one()` is used to configure 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 `set_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.

Now, go back and look at the `dialog_event()` method of this class. All it contains is a case statement that passes the event on to one of several handlers. So, for example, when an event is generated by `check_box_1`, the method `handle_check_box_1(ev)` is invoked. Each handler 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.

Note that any default events, which are those not turned into events by one of the objects, are sent to the `handle_default()` method. This 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.

Containers

Containers are components that contain other components. In fact, the `_Dialog` class itself is in fact a container. We have already seen the `Panel` class in action in the last example. Now let's look at two more useful container objects in the standard toolkit: `TabSet` and `OverlaySet`.

`TabSet`

This class provides a container object that contains several tabbed panes, any one of which is displayed at any given time. The user switches between panes by clicking on the 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 17−3 presents a simple example of a `TabSet` that contains three `TabItems`, each of which contains a single label.

Listing 17−3
TabSet Program

```
#
# Simple example of a TabSet
#
class TabDemo : _Dialog(quit_button)
   #
   # Quit on button press.
   #
   method handle_quit_button(ev)
      if ev.get_code() > 0 then
         dispose()
   end

   method handle_default(ev)
   end

   method dialog_event(ev)
      case ev.get_component() of {
         quit_button : handle_quit_button(ev)
         default : handle_default(ev)
         }
   end

initially
   local tab_set, tab_item_1, label_1, tab_item_2, label_2,
         tab_item_3, label_3

   self._Dialog.initially()
   set_attribs("size=355,295", "font=sans",
               "bg=light gray", "resize=on")

   #
```

```
   # Create the TabSet
   #
   tab_set := TabSet("pos=50%,47", "size=200,150", "align=c,t")

   #
   # First pane
   #
   tab_item_1 := TabItem("label=Pane 1")
   label_1 := Label("pos=50%,50%", "align=c,c", "label=Label 1")
   tab_item_1.add(label_1)
   tab_set.add(tab_item_1)

   #
   # Second pane
   #
   tab_item_2 := TabItem("label=Pane 2")
   label_2 := Label("pos=50%,50%", "align=c,c", "label=Label 2")
   tab_item_2.add(label_2)
   tab_set.add(tab_item_2)

   #
   # Third pane
   #
   tab_item_3 := TabItem("label=Pane 3")
   label_3 := Label("pos=50%,50%", "align=c,c", "label=Label 3")
   tab_item_3.add(label_3)
   tab_set.add(tab_item_3)
   tab_set.set_which_one(tab_item_1)
   add(tab_set)

   #
   # Add a quit button
   #
   quit_button := TextButton("pos=50%,234", "align=c,t",
                             "label=Quit")
   add(quit_button)

   show()
end

procedure main()
   TabDemo()
end
```

The resulting window is shown in Figure 17−3:

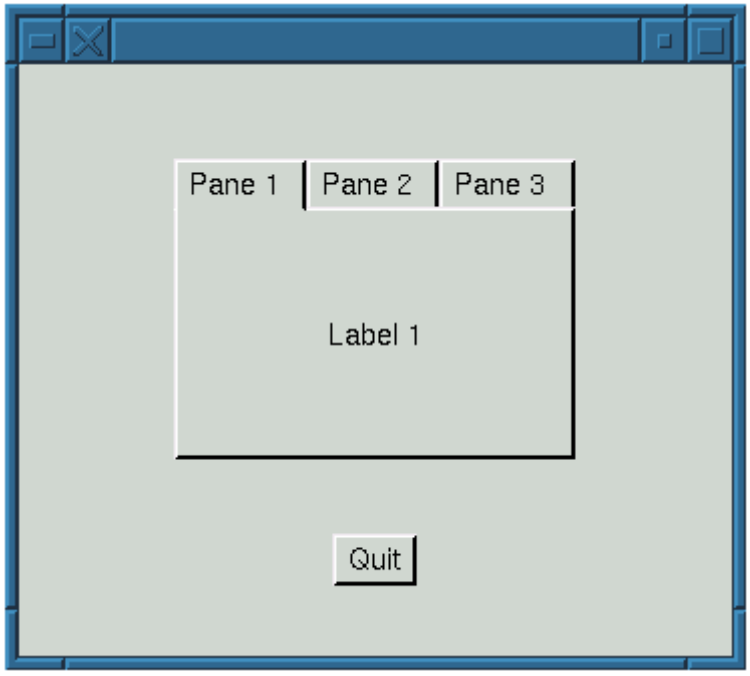

Figure 17−3:
TabSet example window

`OverlaySet`

This class is very similar to the `TabSet` class, but control over which pane is currently on display is entirely under the programmer's control. There is no line of tabs to click.

Instead of adding items to `TabItem` structures, `OverlayItem` objects are used. An empty `OverlayItem` can be used if the area should be blank at some point. The current `OverlayItem` on display is set by the method `set_which_one(x)`, where `x` is the desired `OverlayItem`.

Menu Structures

The toolkit provides all the standard building blocks required to create a menu system. Customized components can also be added, and this is discussed later. The standard components provided are shown in Table 17−1.

Table 17−1
Standard Menu System Components

Component	Description
`MenuBar`	This is the menu area along the top of the window, containing one or more `Menus`.
`MenuButton`	A floating menu bar containing one `Menu`.
`Menu`	A drop down menu pane containing other `Menus` or menu components.
`TextMenuItem`	A textual menu item.

CheckBoxMenuItem	A checkbox in a menu which can be part of a CheckBoxGroup if desired.
MenuSeparator	A vertical separation line between items.

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 17−4 shows our next example program.

Listing 17−4

A Menu Example Program

```
#
# Menu example program.
#
class MenuDialog : _Dialog(text_menu_item_1)
   #
   # Just test for the quit menu option
   #
   method dialog_event(ev)
      if ev.get_component() === text_menu_item_1 then
         dispose()
   end

initially
   local menu_bar, menu_1, menu_2, menu_separator_1, menu_6,
      menu_3, menu_4, menu_5, menu_button_1, menu_7,
      text_menu_item_2, text_menu_item_3, text_menu_item_4,
      text_menu_item_5, text_menu_item_6, text_menu_item_7,
      check_box_menu_item_1, check_box_menu_item_2,
      check_box_menu_item_3, check_box_menu_item_4,
      text_menu_item_8, text_menu_item_9, check_box_group_1

   self._Dialog.initially()
   set_attribs("size=426,270", "font=sans", "bg=light gray")

   check_box_group_1 := CheckBoxGroup()

   #
   # Create the menu bar.  The position and size default to
   # give a bar covering the top of the window.
   #
   menu_bar := MenuBar()

   #
   # The first menu ("File") − just contains one text item.
   #
   menu_1 := Menu("label=File")
   text_menu_item_1 := TextMenuItem("label=Quit")
   menu_1.add(text_menu_item_1)
   menu_bar.add(menu_1)

   #
   # The second menu ("Labels") − add some labels
   #
   menu_2 := Menu("label=Labels")
   text_menu_item_2 := TextMenuItem("label=One")
   menu_2.add(text_menu_item_2)
   text_menu_item_3 := TextMenuItem("label=Two")
   text_menu_item_3.set_label_left("ABC")
   menu_2.add(text_menu_item_3)
   #
   # A separator
   menu_separator_1 := MenuSeparator()
   menu_2.add(menu_separator_1)
```

```
   text_menu_item_4 := TextMenuItem("label=Three")
   text_menu_item_4.set_label_right("123")
   menu_2.add(text_menu_item_4)
   #
   # A sub-menu in this menu, labelled "Images"
   menu_6 := Menu("label=Images")
   #
   # Add three text items with custom images. The rather
   # unwieldy strings create a triangle, a circle and a
   # rectangle.
   #
   text_menu_item_5 := TextMenuItem("label=One")
   text_menu_item_5.set_img_left("15,c1,6666666066666
                                          6666_
   6666606666666666666006666666666660006666666666006006666666_

66600600666666666006600666666666006600666666600666660066666_

60066666006666600666666600666600666666600666006666666600660_
     00000000000600000000000000")
   menu_6.add(text_menu_item_5)
   text_menu_item_6 := TextMenuItem("label=Two")
   text_menu_item_6.set_img_left("15,c1,66666600066666666_

66000000066666600066666000666600666666600666006666666600660_

66666666660600666666666660000666666666660000666666666660060_

66666666660600666666666600666006666666006660006666600666_
     66000000066666666666000666666")
   menu_6.add(text_menu_item_6)
   text_menu_item_7 := TextMenuItem("label=Three")
   text_menu_item_7.set_img_left("15,c1,00000000000000000_

00000000000000066666666660000666666666660000666666666660000_

66666666660000666666666660000666666666660000666666666660000_

66666666660000666666666660000666666666660000666666666660000_
     00000000000000000000000000000")
   menu_6.add(text_menu_item_7)
   menu_2.add(menu_6)
   menu_bar.add(menu_2)

   #
   # The third menu ("Checkboxes")
   #
   menu_3 := Menu("label=Checkboxes")
   #
   # Sub-menu "Group" - two checkboxes in a checkbox group.
   #
   menu_4 := Menu("label=Group")
   check_box_menu_item_1 := CheckBoxMenuItem("label=One")
   check_box_group_1.add(check_box_menu_item_1)
   menu_4.add(check_box_menu_item_1)
   check_box_menu_item_2 := CheckBoxMenuItem("label=Two")
   check_box_group_1.add(check_box_menu_item_2)
   menu_4.add(check_box_menu_item_2)
   menu_3.add(menu_4)
   #
   # Sub-menu - "Alone" - two checkboxes on their own
   #
   menu_5 := Menu("label=Alone")
   check_box_menu_item_3 := CheckBoxMenuItem("label=Three")
   menu_5.add(check_box_menu_item_3)
```

```
   check_box_menu_item_4 := CheckBoxMenuItem("label=Four")
   menu_5.add(check_box_menu_item_4)
   menu_3.add(menu_5)
   menu_bar.add(menu_3)
   add(menu_bar)

   #
   # Finally, create a menu button - a mini floating menu
   # with one menu inside it.
   #
   menu_button_1 := MenuButton("pos=347,50%", "align=c,c")
   #
   # This is the menu, its label appears on the button.  It
   # just contains a couple of text items for illustration
   # purposes.
   #
   menu_7 := Menu("label=Click")
   text_menu_item_8 := TextMenuItem("label=One")
   menu_7.add(text_menu_item_8)
   text_menu_item_9 := TextMenuItem("label=Two")
   menu_7.add(text_menu_item_9)
   menu_button_1.set_menu(menu_7)
   add(menu_button_1)
   show()
end

procedure main()
   MenuDialog()
end
```

The output of this program with the middle menu and its submenu open appears in Figure 17−4.

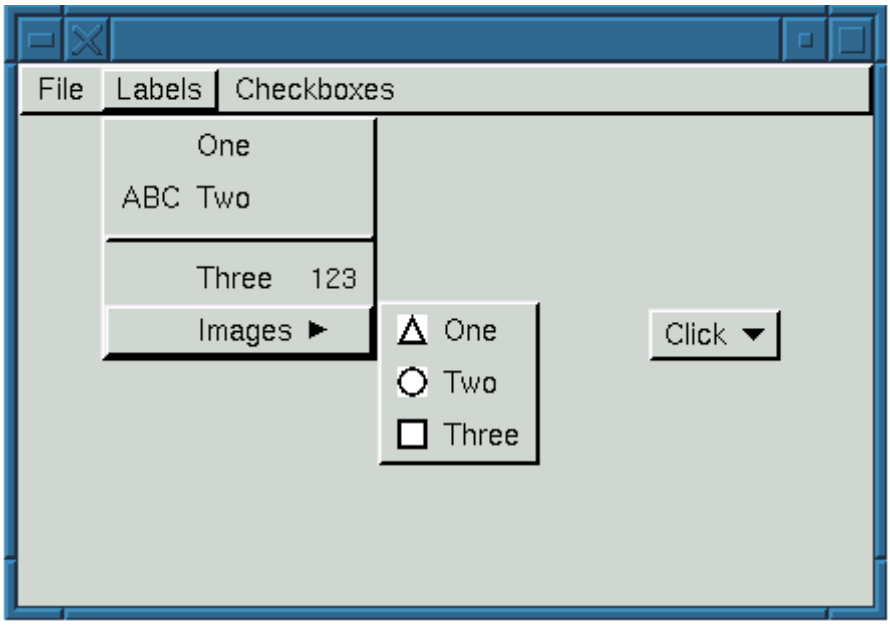

Figure 17−4:
Menus

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 B.

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 `add(c)` method is used to set the title. Here is a program fragment to create a border with a label as its title:

```
b := Border()
#
# Add a Label as the title
#
l := Label("label=Title String")
b.add(l)
add(b)
```

One useful technique is to place a border around a `Panel` object. This is easily accomplished by placing the border within the panel, and making it the same size as the panel by using percentage size specifications:

```
p := Panel()
 ...
b := Border("pos=0,0", "size=100%,100%")
p.add(b)
```

Images and Icons

The toolkit provides support for both images loaded from GIF files and bitmap icons defined by Icon strings. GIF files are manipulated using the Image class. The method `set_filename(x)` is used to set the location of the file to be displayed. The image will be 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.

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. It also contains two *abstract* methods that the new component class must implement itself: one to display the component (`display()`) and one to handle events (`handle_event()`).

A full list of the methods and variables defined in the `Component` class is given in the GUI class library reference in Appendix B. Please refer to that section when reading this next example program, which creates a custom component that implements a simple percentage "progress bar." To give the component something to do with events, when a mouse click occurs in the component, that sets the percentage to the position in the component clicked. The component also extends the resize method. This is simply a convenient point to set a default height and initialize some variables.

Listing 17−5
A ProgressBar Program

```
#
# Include some standard constants.
#
$include "guiconsts.iol"

#
# A custom component − a progress bar
#
class ProgressBar : Component(
   p,              # The percentage on display.
   bar_x, bar_y,
   bar_h, bar_w    # Maximum bar height and width.
   )

   #
   # Resize method − called once at startup and every time the
   # window is resized.
   #
   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 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

   #
```

```
   # The display method.  We use double-buffering to avoid any
   # flicker.  This just means drawing to cbwin and then
   # copying (if buffer_flag is &null).
   #
   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)
      return
   end

   #
   # Handle the event e; if it's a press in the component's area
   # then re-compute the percentage and return an Event.
   #
   method handle_event(e)
      if integer(e) = (&lpress | &rpress | &mpress) &
         in_region() & &x <= bar_x + bar_w then {
         set_percentage((100 * (&x - bar_x)) / bar_w)
         return _Event(e, self, 0)
         }
   end

   #
   # Get the current percentage.
   #
   method get_percentage()
      return p
   end

   #
   # Set the percentage.  The redisplay() method will update the
   # screen if needed by calling display().
   #
   method set_percentage(pct)
      p := pct
      redisplay()
   end

initially
   self.Component.initially()
   # Initialize the percentage.
   set_percentage(0)
end

#
# Very simple dialog to test the above component.
#
class TestProgressBar : _Dialog(pb, close)
   method dialog_event(ev)
```

```
      case ev.get_component() of {
         pb :
            write("pb produced an event - percentage = ",
                  pb.get_percentage())
         close :
            if ev.get_code() > 0 then
               dispose()
      }
   end

initially
   self._Dialog.initially()
   set_attribs("size=400,200", "font=sans", "bg=light gray",
                     "resize=on")

   #
   # Add a progress bar.
   #
   pb := ProgressBar("pos=50%,33%", "size=50%", "align=c,c")
   add(pb)
   #
   # Add a close button.
   #
   close := TextButton("label=Close", "pos=50%,66%", "align=c,c")
   add(close)
   show()
end

#
# Program entry point; just create the dialog.
#
procedure main()
   TestProgressBar()
end
```

The window produced by this program is shown in Figure 17−5.

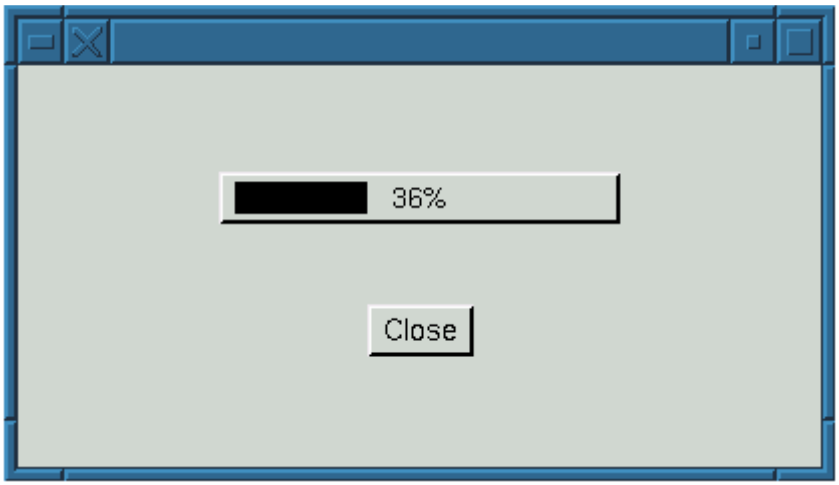

Figure 17−5:
Progress bar program

More complex components use other components within themselves. For example, the `TextList` class contains two `ScrollBars`, each of which in turn contains two `IconButtons`. In this sort of component, certain methods must be extended so that the sub−components are properly initialized.

Figure 17−6 shows a dialog window containing another example custom 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.

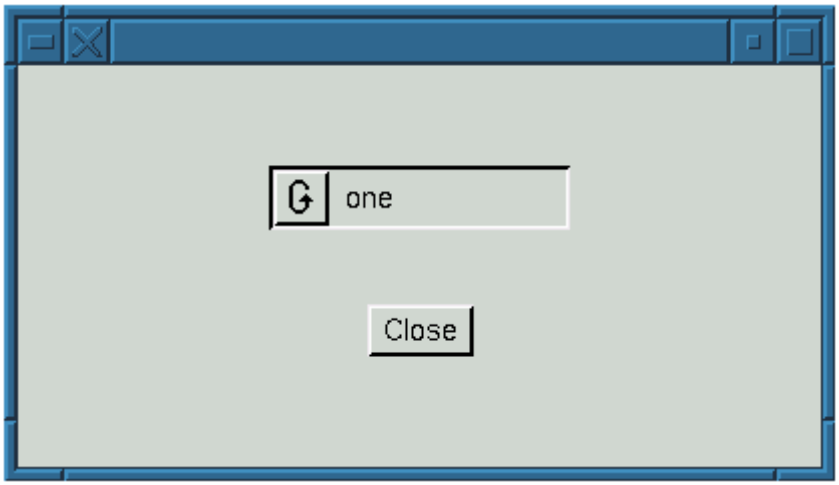

Figure 17−6:
Circulate dialog

In this example, the methods `firstly()`, `finally()` and `set_parent_Dialog()` have been extended to initialize the subcomponents `b` and `l` correctly, and to dispose of their resources when the dialog is closed. The `resize()` method sets the sub−components' position and size, and then calls the `resize()` method for each of them. The `display()` method and `handle_event()` methods are very simple; the latter passes the event on to the button. If it has been pressed, then the label is amended and an event is returned.

Finally, the `final_setup()` method is extended to setup the label and button's `parent_Dialog` and reference variables. By setting the reference to `self`, the two components' position, attributes, and size are computed relative to the `Circulate` object itself. This ensures that if, for example, you added the line `c.set_attribs("bg=red")` to the `initially` section, then both button and label would have this attribute as well as the object itself.

Listing 17−6
Circulate Dialog

```
#
# Include some standard constants
#
$include "guiconsts.iol"

#
# Selection from a list
#
class Circulate : Component(selection, selection_list, b, l)
   #
   # Some methods to set/get the selection data.
   #

   method set_selection_list(x)
```

```
      selection_list := x
      set_selection(1)
      return x
   end

   method set_selection(x)
      selection := x
      l.set_label(selection_list[selection])
      redisplay()
      return x
   end

   method get_selection()
      return selection
   end

   #
   # This method is called after resize, but before the object
has
   # been displayed.
   #
   method firstly()
      self.Component.firstly()
      b.firstly()
      l.firstly()
   end

   #
   # Called after the dialog is closed.
   #
   method finally()
      self.Component.finally()
      b.finally()
      l.finally()
   end

   #
   # Called once at startup, and whenever the window is resized.
   #
   method resize()
      /h_spec := WAttrib(cwin, "fheight") + 16
      self.Component.resize()

      #
      # 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

   #
   # Event handler − if the button is clicked move the selection
   # on one and produce and Event, which is the received by the
   # dialog's dialog_event method.
   #
   method handle_event(e)
      if b.handle_event(e).get_code() > 0 then {
         set_selection(1 + selection % *selection_list)
         return _Event(e, self, 0)
         }
   end

   #
   # This method has to be extended to initialize the label and
   # the button.
   #
   method final_setup(x, y)
      self.Component.final_setup(x, y)
      b.final_setup(x, self)
      l.final_setup(x, self)
   end

initially
   self.Component.initially()
   l := Label()
   l.clear_draw_border()
   b := IconButton()
   b.set_draw_border()
   b.set_img("13,c1,_
     ~~~~0000~~~~~_
     ~~~000000~~~~_
     ~~00~~~~00~~~_
     ~00~~~~~~00~~_
     ~00~~~~~~00~~_
     ~00~~~~~~~~~~_
     ~00~~~~~~~~~~_
     ~00~~~~~~0~~~_
     ~00~~~~~000~~_
     ~00~~~~00000~_
     ~00~~~0000000_
     ~00~~~~~~00~~_
     ~00~~~~~~00~~_
     ~00~~~~~~00~~_
     ~~00~~~~00~~~_
     ~~~000000~~~~_
     ~~~~0000~~~~~_
     ")
end
class TestCirculate : _Dialog(c, close)
   method dialog_event(ev)
      case ev.get_component() of {
         c :
            write("c produced an event − selection = ",
                  c.get_selection())
         close :
            if ev.get_code() > 0 then
               dispose()
         }
```

```
   end

initially
   self._Dialog.initially()
   set_attribs("size=400,200", "bg=light gray",
                     "font=sans", "resize=on")
   c := Circulate("pos=50%,33%", "size=150", "align=c,c")
   c.set_selection_list(["one", "two", "three", "four", "five"])
   add(c)

   #
   # Add a close button.
   #
   clos := TextButton("label=Close", "pos=50%,66%", "align=c,c")
   add(clos)

   show()
end


procedure main()
   TestCirculate()
end
```

Customized menu components

Listing 17−7 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 17−7
Color Palette Program

```
#
# The standard constants
#
$include "guiconsts.iol"

#
# Width of one colour cell in pixels
#
$define CELL_WIDTH 30

#
# The Palette class
#
class Palette : SubMenu(
   colour,                 # Colour number selected
   palette,                # List of colours
   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 any sizes.
   #
   method resize()
      w := h := box_size * CELL_WIDTH + 2 * BORDER_WIDTH
   end

   #
   # Called to display the item.  The x, y coordinates
   # 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_menu_bar.get_parent_win(),
            temp_win, x, y, w, h, 0, 0)
         }

      cw := Clone(parent_menu_bar.cwin)

      #
      # Clear area and draw rectangle around whole
      #
      EraseRectangle(cw, x, y, w, h)
      DrawRaisedRectangle(cw, x, y, w, h)

      #
      # Draw the colour grid.
      #
      y1 := y + BORDER_WIDTH
      e := create "fg=" || !palette
      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.  We return one of three
   # codes depending on whether we have a result (SUCCEED), we
   # have a non-result (a mouse event outside our region: FAIL_2)
   # or neither a definite success or failure (CONTINUE).
   #
   method handle_event(e)
      if i := in_palette_region() then {
         if integer(e) = (&lrelease|&rrelease|&mrelease) then {
            colour := i
            return MenuEvent(SUCCEED, e, self, 0)
            }
         }
      else {
         if integer(e) = (&lrelease | &rrelease | &mrelease |
                          &lpress | &rpress | &mpress) then
            return MenuEvent(FAIL_2)
         }
      return MenuEvent(CONTINUE)
   end

   #
   # Close this menu.
   #
   method hide()
      #
      # Restore window area.
      #
      cw := parent_menu_bar.cwin
      EraseRectangle(cw, x, y, w, h)
      CopyArea(temp_win, parent_menu_bar.get_parent_win(), 0, 0,
               w, h, x, y)
      WClose(temp_win)
      temp_win := &null
   end

initially
      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.
      #
      set_img_right(img_style("arrow_right"))
end

#
# Test class dialog.
#
class TestPalette : _Dialog(text_menu_item, palette, close)
   method dialog_event(ev)
      case ev.get_component() of {
         palette :
            write("Colour selected : " || palette.get_colour())
         close :
            if ev.get_code() > 0 then
               dispose()
         }
   end
initially
   local menu_bar, menu
   self._Dialog.initially()
```

```
   set_attribs("size=400,200", "bg=light gray",
                     "font=sans", "resize=on")

   #
   # Create a MenuBar structure which includes our palette as a
   # submenu
   #
   menu_bar := MenuBar("pos=0,0")
   menu := Menu("label=Test")
   text_menu_item := TextMenuItem("label=Anything")
   menu.add(text_menu_item)
   palette := Palette("label=Test menu")
   palette.set_palette(["red", "green", "yellow", "black",
                        "white","purple","gray","blue","pink"])
   menu.add(palette)
   menu_bar.add(menu)
   add(menu_bar)

   #
   # Add a close button.
   #
   clos := TextButton("label=Close", "pos=50%,66%", "align=c,c")
   add(clos)

   show()
end

#
# Main program entry point.
#
procedure main()
   TestPalette()
end
```

The resulting window, with the `Palette` menu active is shown in Figure 17−7.

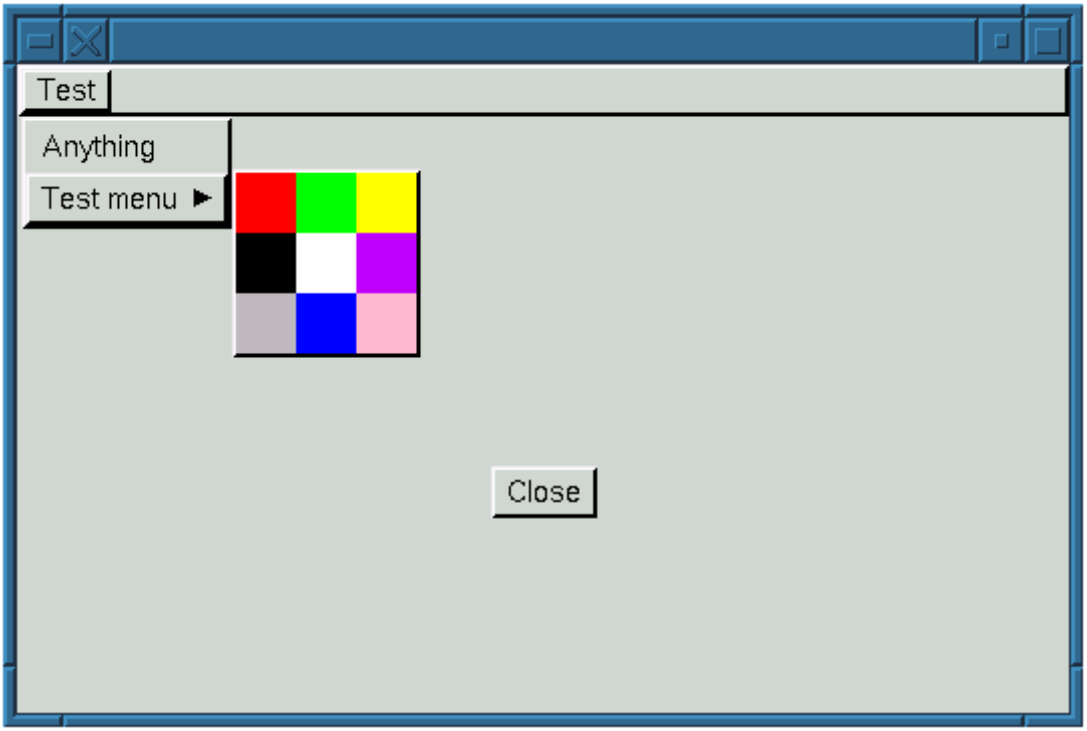

Figure 17−7:
TestPalette window

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 sub−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. This is very easy to do by following the following structure in the dialog class:

```
class AnyDialog : _Dialog(a_variable)
   method set_a_variable(x)
      a_variable := x
   end
   ...
   initially
      self.Component.initially()
      ...
      # Initialize Components, but DON'T
      # call the show method!
end
```

You then use the following code in the calling program:

```
procedure call_any_dialog()
   local d
   d := AnyDialog()
   d.set_a_variable("something")
   d.show()
end
```

However, what if you want to set the variables before you create the components in the dialog? You might want to do this, for example, if you want to initialize the contents of a label to a parameter from the calling program. The solution is to override the `show()` method and put the dialog initialization there. The dialog then has the following structure:

```
class AnyDialog : _Dialog(a_variable)
   method set_a_variable(x)
      a_variable := x
   end
   ...
   method show()
      # Initialize the Components.
      ...
      self._Dialog.show()
   end

initially
   self.Component.initially()
   # Some initializing can be done here if desired,
   # or the initially method can be omitted entirely.
   ...
end
```

The calling program still takes the same form.

Getting results back out to the calling program is very easy. The dialog can just set a result variable that can be retrieved by the caller using one of the dialog's methods. To give an example of a dialog window that takes parameters in and passes them out, Figure 17−8 shows an example of a `FileDialog` class for selecting a file.

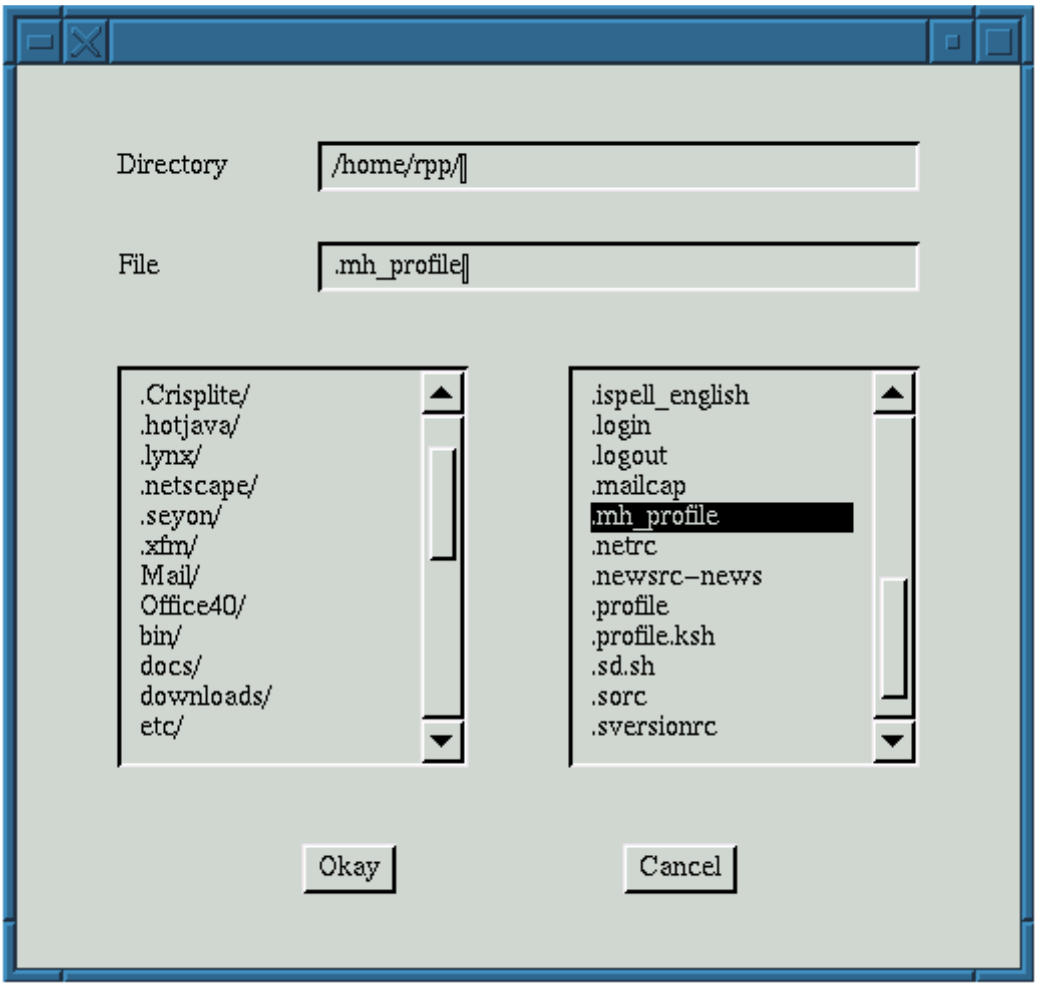

Figure 17−8:
FileDialog window

The source code is shown in Listing 17−8.

Listing 17−8
File Selection Dialog Program

```
#
# FileDialog class
#
class FileDialog : _Dialog(
   init_dir,                # Initial directory name
   init_file,               # Initial file name
   res,                     # Resulting file path
   dir,                     # TextField directory
   file,                    # TextField filename
   dlist,                   # TextList of directories
   flist,                   # TextList of files
   okay,                    #
   cancel,                  #
```

```
extra_attribs            # Custom attributes
)

#
# Extra attributes set by caller.
#
method set_extra_attribs(l)
   return extra_attribs := l
end

#
# Get the directory part of the result
#
method get_directory()
   return directory_name(\res)
end

#
# Get the result, (will fail if cancel was pressed).
#
method get_result()
   return \res
end

#
# Set the initial directory.
#
method set_directory(s)
   return init_dir := s
end

#
# Set the initial file
#
method set_file(s)
   return init_file := s
end

#
# Set the initial file/directory from a whole path.
#
method set_path(s)
   init_dir := directory_name(s)
   init_file := file_name(s)
   return s
end

#
# Set the result
#
method set_result()
   res := get_std_dir() || file.get_contents()
end

#
# Get the directory TextField contents standardized with a
# trailing /
#
method get_std_dir()
   s := dir.get_contents()
   if (s[-1] ~== "/") then
      s ||:= "/"

   return s
end
```

```
method dialog_event(ev)
   case ev.get_component() of {
      cancel :
         if ev.get_code() > 0 then
            dispose()
      okay :
         if ev.get_code() > 0 then {
            set_result()
            dispose()
            }
      file :
         #
         # If return pressed in file TextField, same as okay
         #
         if ev.get_code() = 0 then {
            set_result()
            dispose()
            }
      dlist : {
         #
         # Clicked in the directory list; get the item clicked
         # on.
         #
         value := dlist.get_contents()[
                  dlist.get_selections()[1]]
         s := get_std_dir()
         #
         # Go to parent directory (unless at root directory)
         #
         if value == "../" then {
            if s ~== "/" then {
               s[-1] := ""
               while s[-1] ~== "/" do s[-1] := ""
               }
            dir.set_contents(s)
            }
         else
            dir.set_contents(s ||:= value)
         #
         # Update directory and file lists.
         #
         l1 := []
         l2 := []
         get_directory_list(s, l1, l2)
         dlist.set_contents(l1)
         flist.set_contents(l2)
         file.set_contents("")
         }

      flist :
         #
         # Clicked in file list; set TextField
         #
         file.set_contents(
            flist.get_contents()[flist.get_selections()[1]])

      dir : {
         if ev.get_code() = 0 then {
            #
            # Return pressed in directory TextField; update
            # lists.
            #
            l1 := []
            l2 := []
```

```
            dir.set_contents(s := get_std_dir())
            get_directory_list(s, l1, l2)
            dlist.set_contents(l1)
            flist.set_contents(l2)
            file.set_contents("")
            }
         }
      }
end

method show()
   #
   # Defaults if none set by caller.
   #
   if /init_dir | (init_dir == "") then {
      if p := open("pwd", "pr") then {
         init_dir := read(p)
         close(p)
         }
      }
   /init_dir := "/"
   /init_file := ""

   if (init_dir[-1] ~== "/") then init_dir ||:= "/"

   set_attribs_list(["size=500,450", "resize=on"] |||
                         extra_attribs)

   l := Label("label=Directory", "pos=50,50", "align=l,c")
   add(l)

   dir := TextField("pos=150,50", "size=100%-200",
                    "align=l,c")
   dir.set_contents(init_dir)
   add(dir)

   l := Label("label=File", "pos=50,100", "align=l,c")
   add(l)

   file := TextField("pos=150,100", "size=100%-200",
                     "align=l,c")
   file.set_contents(init_file)
   add(file)

   l1 := []
   l2 := []
   get_directory_list(init_dir, l1, l2)

   dlist := TextList("pos=50,150", "size=50%-75,100%-250")
   dlist.set_select_one()
   dlist.set_contents(l1)
   add(dlist)

   flist := TextList("pos=50%+25,150", "size=50%-75,100%-250")
   flist.set_select_one()
   flist.set_contents(l2)
   add(flist)

   okay := TextButton("label=Okay", "pos=33%,100%-50",
                      "align=c,c")
   add(okay)

   cancel := TextButton("label=Cancel", "pos=66%,100%-50",
                        "align=c,c")
   add(cancel)
```

```
      set_initial_focus(file)
      self._Dialog.show()
   end

initially
   self._Dialog.initially()
   extra_attribs := []
end

#
# Read a directory.
#
procedure get_directory_list(s, dir_list, file_list)
   p := open("ls -alL 2>/dev/null " || s, "pr") |
      fatal_error("couldn't run ls")
   read(p)
   while s := read(p) do {
      s ? {
         if ="d" then {
            move(54)
            put(dir_list, tab(0) || "/")
            }
         else {
            move(55)
            put(file_list, tab(0))
            }
         }
      }
   return
end

#
# Return the directory name of the file name s, including the
# trailing /
#
procedure directory_name(s)
   local i
   every i := find("/", s)
   return s[1:\i + 1] | ""
end

#
# Return the file name of s
#
procedure file_name(s)
   local i
   every i := find("/", s)
   return s[\i + 1:0] | s
end
```

Subclassing

Sometimes it happens that a program has several dialogs that have several components just the same. Obviously it would be nice to create a parent class that contained these dialogs' common elements. One way to achieve this is to use the following structure:

```
class ParentDialog : _Dialog( ...common components...)
   method dialog_event(ev)
      case ev.get_component() of {
         #
         # Handle any or all common components
         #
      }
   end
```

```
   #
   # Set up common components
   #
   method setup()
      self._Dialog.initially()
      # Create and initialize the common components
      ...
   end
end

class SubDialog : ParentDialog(...sub−dialog’s components...)
   method dialog_event(ev)
      self.Parent.dialog_event(ev)
      #
      # Handle our own components
      #
      ...
   end

initially
   #
   # Call the parent class’s setup method.
   #
   setup()
end
```

The reason the code uses a `setup()` method rather than an `initially` method in the parent class is so that parameters may be added if desired. Obviously this template can be structured in many different ways to suit different situations.

Ivib

Creating a dialog window with many components can be hard work, involving repeated compiles and runs to get the components correctly sized and positioned. Furthermore, much of the code in a dialog is lengthy and tiresome to write. To help reduce the work involved for the programmer in creating a dialog, a visual interface builder is available, called Ivib. This program allows a user to interactively place and configure components in a window area. A program is then generated automatically that implements the interface. Ivib owes inspiration to 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 17−9.

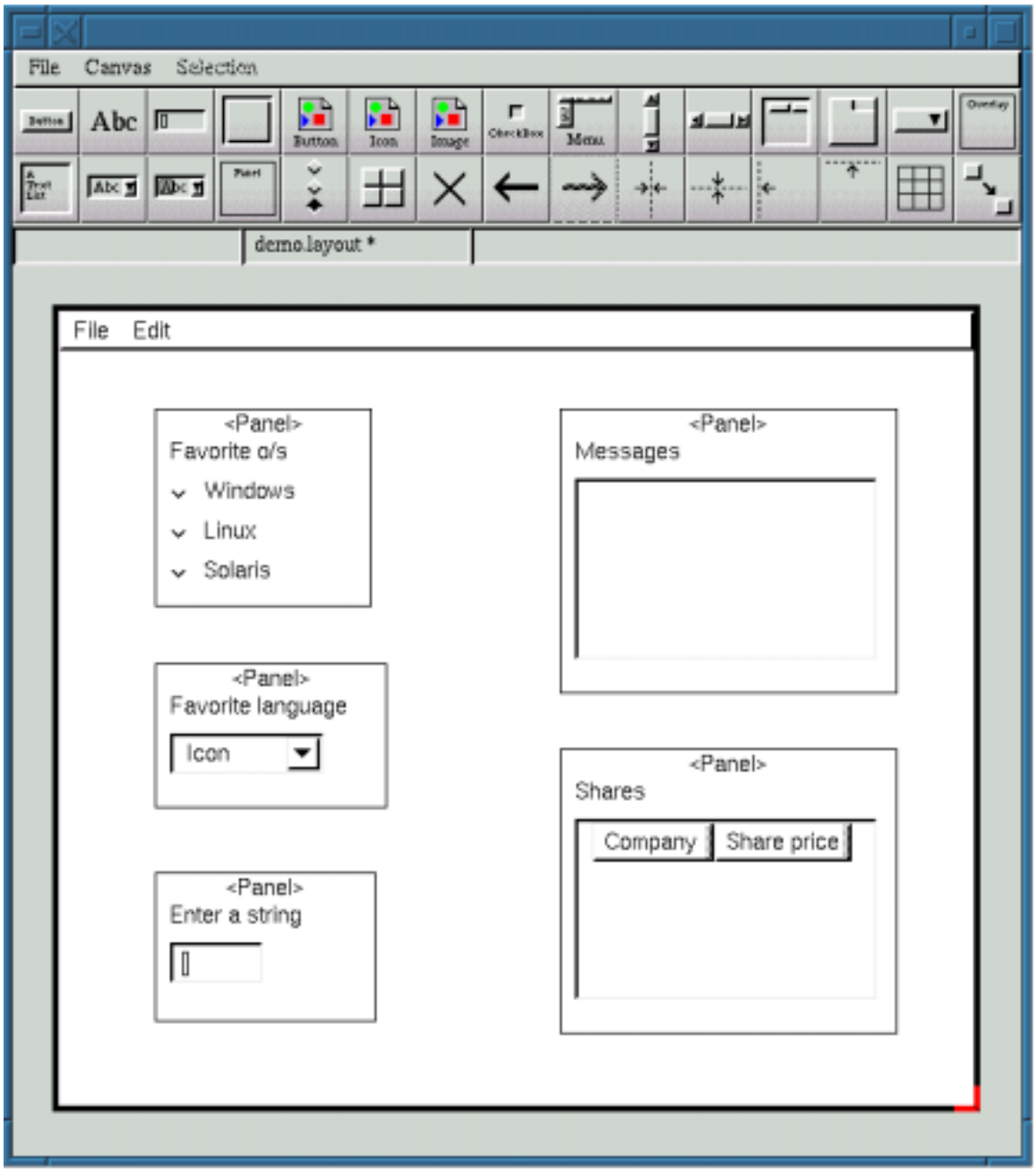

Figure 17−9:
Ivib main window

To create a dialog window using Ivib, start the program with the name of a new layout file. For example:

```
ivib myprog.layout
```

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 17−10 appears.

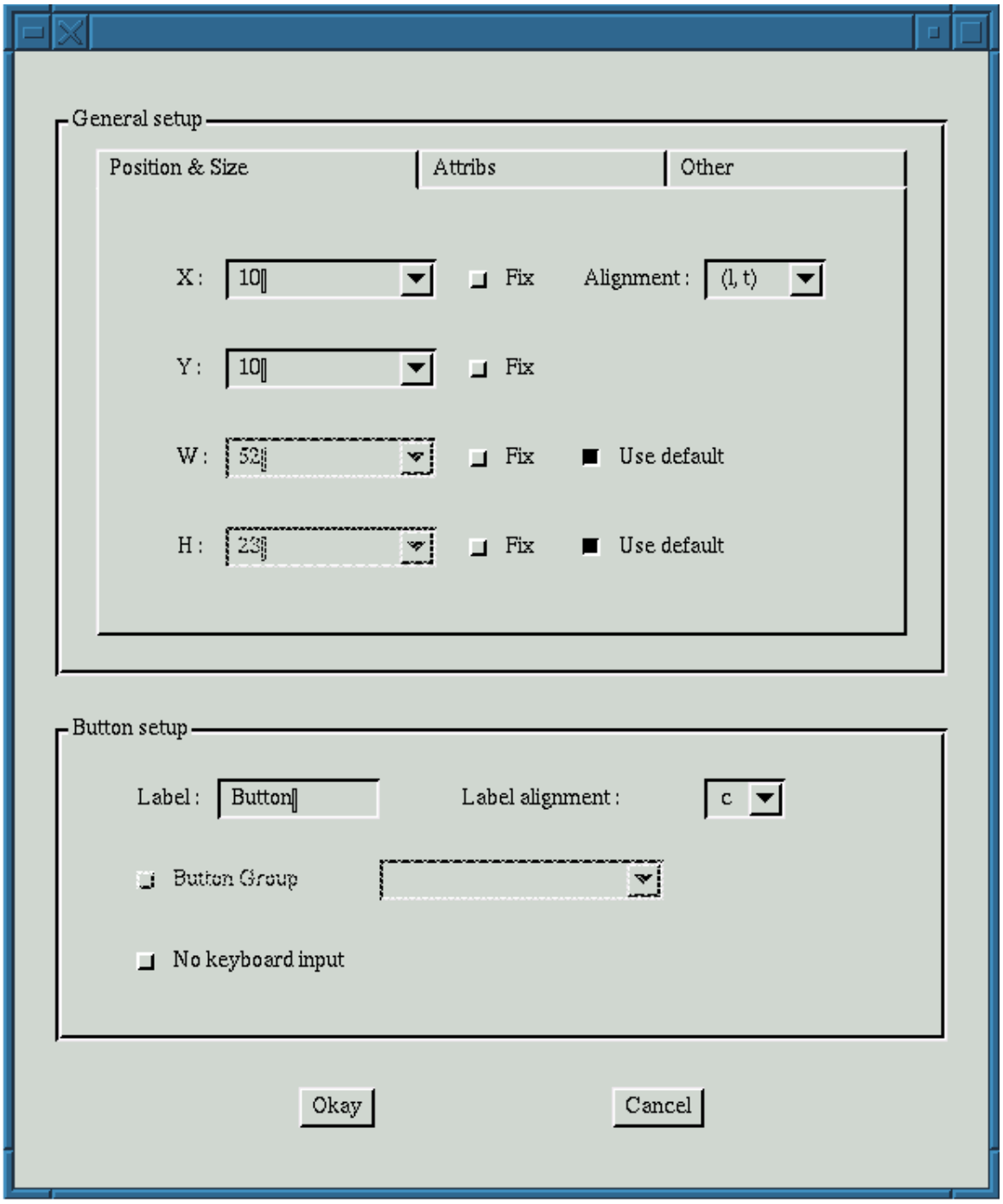

Figure 17−10:
Button configuration window

Change the label by simply changing the text in the "Label" field and clicking "Okay".

As just mentioned, the dialog's attributes are initially the default Icon window attributes. To change these, select the menu option Canvas −> Dialog prefs. The window shown in Figure 17−11 will appear.

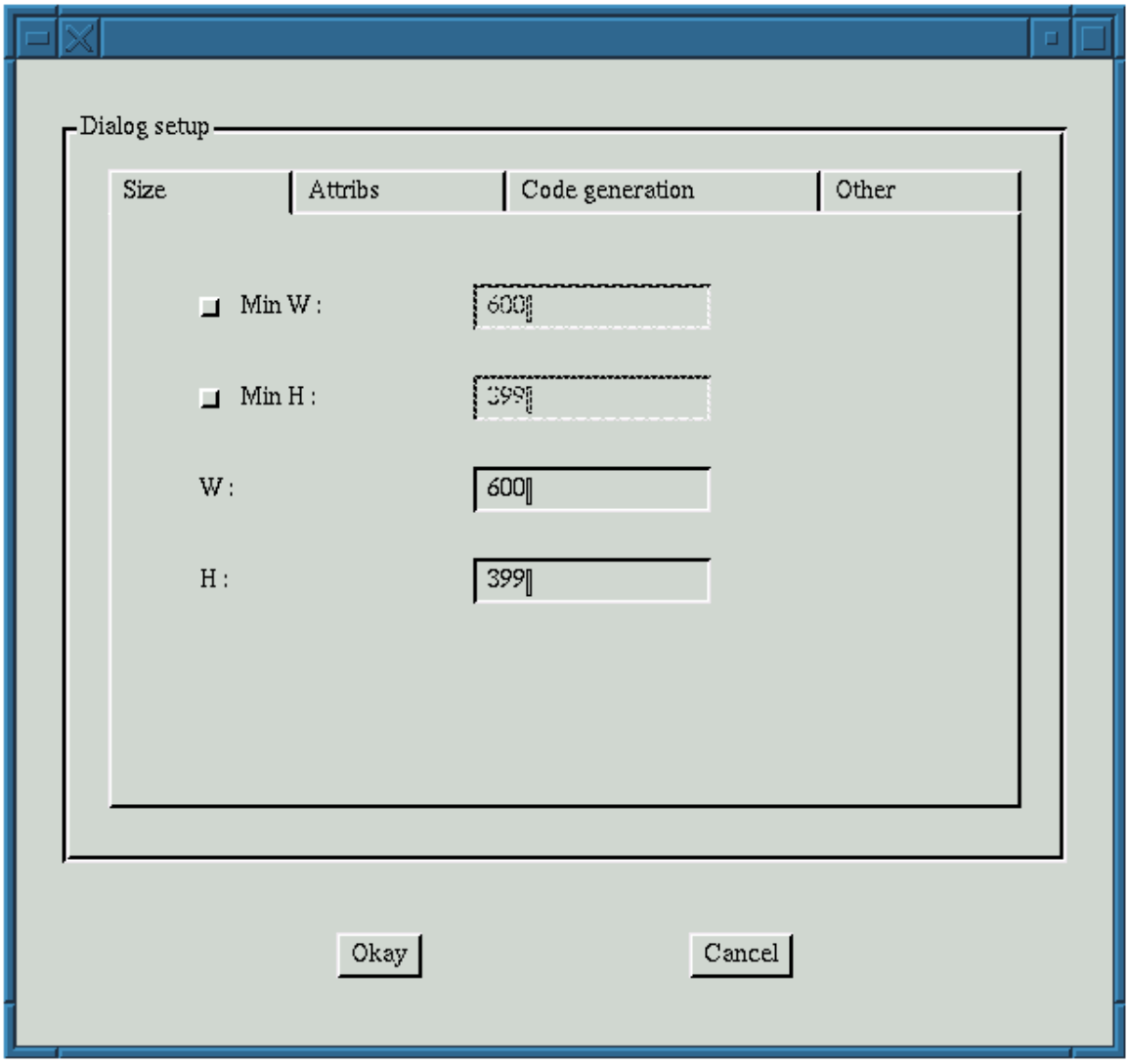

Figure 17−11:
Dialog configuration window

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. Note that a canvas is saved in a .layout file. This is an ASCII file, but it is not really human−readable. Now try creating an Unicon source file. Press Alt−C; again select a filename or accept the default file (note that the file will be overwritten). The extension this time should be `.icn` for a source file. After selection, the program will be output to the given source file. If this is called, for example, `myprog.icn`, then this can be compiled with

```
unicon myprog gui.u
```

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 of how the individual components, dialogs, and operations work.

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. Note that 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

The Other tab allows the name of the object to be set − this is the name used in the output program code. The "Draw Border" button applies to some objects −(see the reference section 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

The Border component is rather odd. To select it, 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 `CheckBoxGroup` may be selected if one is available; this is explained in more detail shortly.

MenuBar

The `MenuBar` dialog enables a complete multi−level menu to be created. Clicking on a particular line will allow an item to be inserted at that point. (Only menus can be inserted into the lowest level of course). 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. Note that 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, and when the dialog is exited, it is this `TabItem` that is to the front of the `TabSet`. To add items to the current pane, simply drag and drop them into it. The whole of the item must be in the pane, and a confirmatory message appears to indicate that the item has been added to the container. To take it out of the container, just 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 just a menu system with one root menu. The dialog is the same as that for `MenuBar` 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 on the screen, but rather places several selected `CheckBox` objects into a `CheckBoxGroup` object, so that they act as coordinated radio buttons. To use this button, select several `CheckBox` objects, and press the button.

The `CheckBoxGroup` itself is configured by selecting the menu item Canvas −> CheckBoxes. In fact, the only attribute to be configured is the name of the `CheckBoxGroup`. Note that 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

This section describes some other editing functions which can be applied to the dialog box 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

The Align horizontally operation sets the X position and the X alignment of all the selected objects to the X position and the X alignment of the first selected object. Note that whether the objects end up appearing to be left aligned, center aligned, 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

The Equalize Widths 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

The Even Space Horizontally operation leaves the leftmost and rightmost object of the current selections in place and moves all of the other objects so that they are equally spaced between the leftmost and rightmost objects. "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.

Summary

The Unicon GUI toolkit offers a full−featured and attractive way of constructing interfaces. The toolkit has modern features − such as tables, tabbed property sheets, and multiline text editing capability − that are not present in Icon's earlier vidgets library. Many components that are present in both libraries are more flexible in the GUI toolkit, supporting fonts, colors, and graphics that the vidgets library does not handle well. The ivib interface builder tool provides programmers with easy access to the GUI toolkit.

The object−orientation of the class library mainly affects its extensibility, although arguably it may also contribute 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.

Chapter 18: Scanning and Parsing

Scanners and parsers are language processing components routinely used by writers of compilers and interpreters. Techniques developed for scanning and parsing may be useful to any application that processes complex input, such as:

- a desktop calculator with arithmetic expression evaluation
- a compiler for a programming language
- an HTML style checker
- a spreadsheet with formulas in the cells

This chapter describes versions of the classic UNIX scanner and parser generators, `lex` and `yacc`, that support Icon. The Icon−friendly tools are called `ilex` and `iyacc`. Note that scanning in this chapter refers to lexical analysis as found in typical compilers, the processing of each character one at a time to form words or tokens. This is not to be confused with scanning an image of a piece of paper, nor is it synonymous with Icon's built−in string scanning mechanism, although string scanning can certainly be used to implement a scanner for a compiler.

Building a scanner or parser can be a daunting task for the newcomer. However, if you start simple, it is very easy to work with these very high−level tools with only a day's worth of experience. For people with experience using the C versions of `lex` and `yacc`, be assured that all the usual features available in those programs are supported here. Extensive and complete documentation on `ilex` and `iyacc` is available on the accompanying CD−ROM. You can check for updated versions of these tools on the Unicon Web page.

When you're finished with this chapter, you will know how to:

- build scanners using `ilex`
- build parsers using `iyacc`
- use the two tools together with Icon's data structures

What are Scanning and Parsing?

Scanning and parsing originate in the field of language recognition. Consider the sentence *This computer can run my software*. Scanning is the recognition of words, the grouping of the characters into contiguous blocks of letters such as *This* and *computer*. The spaces and period are identified as separators during scanning and ignored thereafter.

Parsing is the recognition of valid phrases and sentences. Most sentences in English have a subject phrase and a verb phrase, and optionally an object phrase. The rules that determine what is a valid sentence form a grammar that allows you to organize, understand, and describe sentences. In the above sentence, *This computer* is a subject phrase, *can run* is a verb phrase, and *my software* is an object phrase. In general the modifier comes before the thing modified within the phrases. So *This* modifies *computer*, *can* modifies *run*, and *my* modifies *software*. This hierarchical grouping of words is a by−product of parsing. Figure 18−1 shows a parse tree for the example sentence.

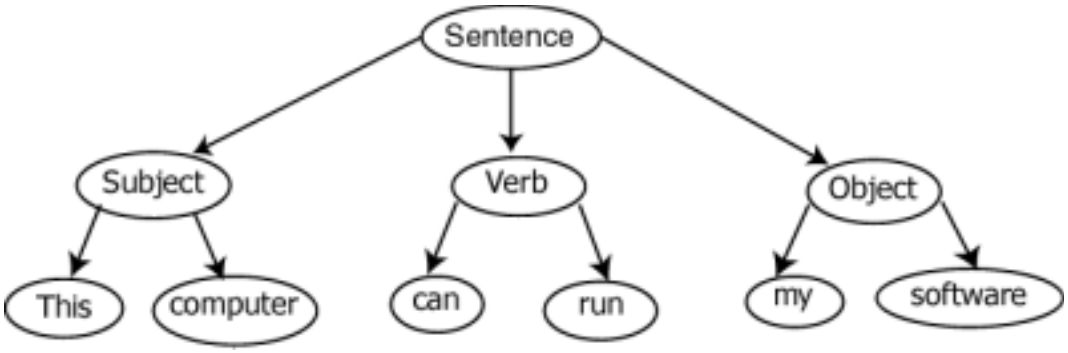

Figure 18−1:
Parse Tree for "This computer can run my software"

If you have studied any linguistics, this description of scanning and parsing may seem nauseatingly oversimplified, and if you have not, then you may be lost already. The tools presented in this chapter can't handle the complexities of human language, but they use basic linguistic concepts to elegantly solve most problems encountered in the processing of computer languages.

This chapter is meant to illustrate the use of `ilex` and `iyacc` in Icon programs. It is not a comprehensive description of these complex tools. If this chapter is your first introduction to `lex` and `yacc`, you may wish to also read "lex & yacc," by John Levine, Tony Mason, and Doug Brown, or consult a reference on writing compilers for more information on these topics.

The ilex Tool

The name of the `ilex` tool stands for Icon's Lexical Analyzer. It was created by modifying a program called `flex` to generate Icon code as an alternative to C. The `flex` program in turn is based on a classic UNIX program called `lex` that dates back to 1975. The `ilex` tool takes a lexical specification and produces a lexical analyzer that corresponds to that specification. A lexical analyzer is a fancy name for a scanner, and the lexical analyzer generated by `ilex` is simply a procedure named `yylex()`.

Chapter 3 describes some library functions that employ regular expressions, a notation for specifying the structure of words. The specification of the lexical structure of many languages can be concisely and precisely stated using this notation. You only need to know the basics of regular expressions to get a useful understanding of them. The `ilex` tool specifications consist of a list of regular expressions.

A Word Count Program

There is a UNIX program called `wc`, short for word count, that counts newlines, words, and characters in a file. In this section, you will see how to build such a program using `ilex`. A short, albeit simplistic, definition of a word is any sequence of non−white space characters. White space characters are blanks and tabs. Below is a complete `ilex` program that operates like `wc`:

```
ws    [ \t]
nonws [^ \t\n]
```

```
%{
global cc, wc, lc
%}

%%

{nonws}+    cc +:= yyleng; wc +:= 1
{ws}+       cc +:= yyleng
\n          lc +:= 1; cc +:= 1

%%

procedure main()
   cc := wc := lc := 0
   yylex()
   write(right(lc, 8), right(wc, 8), right(cc, 8))
end
```

All `ilex` programs, including this program, consist of three sections, separated by lines containing two percent signs. The three sections are the definitions section, the rules section, and the procedures section.

In the word count program, the definitions section has two definitions, one for white space (`ws`) and one for non−white space (`nonws`). These definitions are followed by code to declare three global variables, which will be used as counters. The variables `cc`, `wc`, and `lc` are used to count the characters, words, and lines, respectively.

The rules section in this example contains three rules. White space, words, and newlines each have a rule that matches and counts their occurrences. The procedure section has one procedure, `main()`. It calls the lexical analyzer and then prints out the counts. There are many ways to write this word count program, with different performance characteristics. If speed is your primary consideration you can look in the documentation for `flex` to get five progressively more complex but faster versions of word count. The documentation is available on the Internet, just search for flex and Vern Paxson, the author.

A Lexical Analyzer for a Desktop Calculator

The above example illustrates using `ilex` to write standalone programs, but the function `yylex()` produced by `ilex` is usually called by a parser algorithm. The `yylex()` function can be used to produce a sequence of words, and a parser such as that generated by the `iyacc` program combines those words into sentences. So it makes sense to study how `ilex` is used in this typical context. One obvious difference is that in the earlier example, `yylex()` was only called once to process an entire file; in contrast, when a parser uses `yylex()` it calls it repeatedly, and `yylex()` returns with each word that it finds. You will see this in the following example.

A calculator program is simple enough to understand in one sitting and complex enough to get a sense of how to use `ilex` with its parser generator counterpart, `iyacc`. In general, in a desktop calculator program the user types in complex formulas and the calculator evaluates them and prints out the answer.

First things first: what are the *words* of this little language? Numbers, math operators, and variable names. A number is one or more digits followed by an optional decimal point and one or more digits. In regular expressions, you can write this as

```
[0-9]+(\.[0-9]*)?
```

The 0 through 9 is specified with a range using a dash for characters within the square brackets. The plus sign means one or more occurrences, whereas the star means *zero* or more occurrences. The backslash period means literally match a period; without a backslash a period matches any single character. The parentheses are used for grouping and the question mark means zero or more times.

The math operators are simple "words" composed of one character such as the plus sign, minus sign, and star for multiplication. Variable names need to be meaningful; so why not let them be any combination of letters, digits, and underscores. You do not want to confuse them with numbers, so refine the definition by making sure that variables do not begin with a number. This definition of variable names corresponds to the following regular expression:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Here is some more information about the three sections that make up an `ilex` program. The definitions section contains Icon code that is copied verbatim into the generated final program, before the generated scanner code. You can put `$include` statements there to define symbolic constants for the different kinds of words in your scanner. The rules section contains a series of rules, composed of two parts: a regular expression and a fragment of Icon code that executes whenever that regular expression matches part of the input. The procedures section contains arbitrary code that is copied verbatim after the generated scanner's code. A complete scanner specification for the desktop calculator looks like:

```
%{
# y_tab.icn contains the symbol definitions for integer values
# representing the terminal symbols NAME, NUMBER, and
# ASSIGNMENT. It is generated with iyacc –d calc.y
$include y_tab.icn
%}

letter     [a-zA-Z_]
digiletter [a-zA-Z0-9_]

%%

{letter}{digiletter}*  { yylval := yytext; return NAME }
[0-9]+(\.[0-9]+)?      { yylval := numeric(yytext); return NUMBER
}
\n                      return 0     /* logical EOF */
":="                    return ASSIGNMENT
[ \t]+                  ; /* ignore whitespace */
.                      return ord(yytext)
%%
```

There are a couple of details about `ilex` and `iyacc` worth noting in this scanner. The `ilex` tool maintains a global variable named `yytext` that holds the characters that match a given regular expression. For example, a plus operator is returned by the scanner rule that says to return the character code corresponding to `yytext`, `ord(yytext)`, on the regular expression that consists of a lone period (`.` matches any one character).

Even if `yytext` is not part of `yylex()`'s return value for a token, there are situations in which the string is of interest to the parser, which reads lexical values from a global variable called `yylval`. When a variable name is encountered, it makes sense to copy

`yytext` over into `yylval`. On the other hand, when a number is encountered, the numeric value corresponding to the characters in `yytext` is computed and stored in `yylval`. Since Icon allows a variable to hold any type of value, there is no need for a union or some other messy construct to handle the fact that different tokens have different kinds of lexical values.

A Summary of the Regular Expression Operators:

The above example only uses a few of the regular expression operators. Table 18−1 lists the most commonly used operators in `ilex`. Advanced users will want to consult the documentation for `flex` or its predecessor, `lex`, for a more complete list.

Table 18−1
Commonly used ilex operators

Operator	Description
.	Matches any single character except the newline character.
*	Matches zero or more occurrences of the preceding expression.
[]	This is a character class that matches any character within the brackets. If the first character is a caret (^), then it changes the meaning to match any character *except* those within the brackets. A dash inside the brackets represents a character range, so [0−9] is equivalent to [0123456789].
^	Matches the beginning of a line. This interpretation is used only when the caret is the first character of a regular expression.
$	Matches the end of a line. This interpretation is used only when the dollar symbol is the last character of a regular expression.
\	Used to escape the special meaning of a character. For example, \$ matches the dollar sign, not the end of a line.
+	Matches one or more occurrences of the preceding expression. For example, [0−9]+ matches "1234" or "734" but not the empty string.
?	Matches zero or more occurrences of the preceding expression. For example, −?[0−9]+ matches a number with an optional leading negative sign.
|	Matches either the preceding regular expression or the one following it. So Mar|Apr|May matches any one of these three months.
"..."	Matches everything in quotes literally.
()	Groups a regular expression together, overriding the default operator precedence. This is useful for creating more complex expressions with *, +, and |.

Before you conclude your study of `ilex`, two subtle points are worth knowing. The matches allowed by a list of regular expressions are often ambiguous, and this is normal and healthy. For example, does `count10` match a variable name and then an integer, or just one variable name? The `ilex` tool matches the longest substring of input that can match the regular expression. So it matches `count10` as one word, which is a variable name in this case. There is one more sticky point: what if two rules match the exact same input characters, with no longest match to break the tie? In this case, `ilex` picks the first rule listed in the specification that matches, so the order of the rules can be important.

The iyacc Tool

The `iyacc` program stands for Icon's Yet Another Compiler Compiler. It is a variant of a program called `byacc` (Berkeley YACC), modified to generate Icon code as an alternative to C. The `iyacc` program takes a grammar that you specify and generates a parser that recognizes correct sentences. A parser allows you to do more than just tell whether a sentence is valid according to the grammar, it allows you to record the structure of the sentence for later use. This internal structure explains *why* the sentence is grammatical, and it is also the starting point for most translation tasks. The structure is often a tree called a *parse tree.*

The language used by `iyacc` to express the grammar is based on a form of BNF, which stands for Backus−Naur Form. A grammar in BNF consists of a series of production rules, each one specifying how a component of the parse is constructed from simpler parts. For example, here is a grammar similar to the one used to build the calculator:

```
assignment : NAME := expression
expression : NUMBER
           | expression + NUMBER
           | expression – NUMBER
```

This grammar has two rules. The symbol to the left of the colon is called a *non−terminal* symbol. This means that the symbol is an abstraction for a larger set of symbols. Each symbol to the right of the colon can be either another non−terminal or a *terminal* symbol. If it is a terminal, then the scanner will recognize the symbol. If it is not, then a rule will be used to match that non−terminal. It is not legal to write a rule with a terminal to the left of the colon. The vertical bar means there are different possible matches for the same non−terminal. For example, an expression can be a number, an expression plus a number, or an expression minus a number.

As was mentioned previously, the most common way to represent the result of parsing is a tree. For example, if the input string is `"count := 10 + 99"`, then the parse tree for the above grammar would look like Figure 18−2. Note how the terminal symbols NAME and NUMBER have lexical values provided by the scanner.

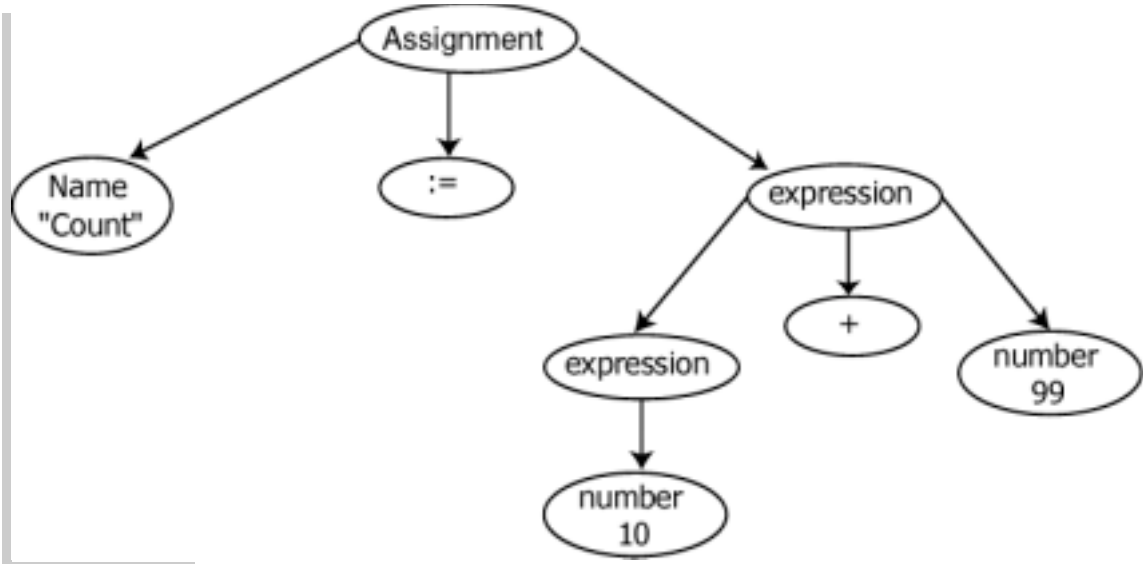

Figure 18−2:
Parse Tree for "count := 10 + 99"

The `iyacc` program's input files are structured much like `ilex`'s input files. They have a definitions section, a rules section, and a procedures section. The definitions section includes code to be copied into the output file before the parser code, surrounded by `%{` and `%}`. This section also includes the declarations of each of the *tokens* that are going to be returned by the scanner. The term "token" is just another name for a terminal symbol, or a word. Tokens can be given a left or right associativity, and they are declared from lowest precedence to highest precedence. You may want to consult a reference on `yacc` for details on how these features are used.

The next section is the rules section. The left and right sides of a rule are separated by a colon. The right side of the rule can be embedded with semantic actions consisting of Icon code that is surrounded by curly braces. Semantic actions execute when the corresponding grammar rule is matched by the input. The last section is the procedures section. In the calculator, there is one procedure, `main()`, that calls the parser once for each line. Listing 18−1 shows a complete parser for the desktop calculator.

Listing 18−1
The desktop calculator in calc.y

```
%{

## add any special linking stuff here
global vars

%}

/* YACC Declarations */
%token NUM, NAME, ASSIGNMENT
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */

/* Grammar follows */
%%
input:    /* empty string */
     | input line
     ;

line:     '\n'
    | exp '\n'  { write($1) }
    | NAME ASSIGNMENT exp '\n' {
         vars[$1] := $3
         write($3)
         }
    ;

exp: NUM                 { $$ := $1       }
   | NAME              { $$ := vars[$1] }
   | exp '+' exp       { $$ := $1 + $3  }
   | exp '-' exp       { $$ := $1 - $3  }
   | exp '*' exp       { $$ := $1 * $3  }
   | exp '/' exp       { $$ := $1 / $3  }
   | '-' exp  %prec NEG { $$ := -$2      }
   | exp '^' exp       { $$ := $1 ^ $3  }
   | '(' exp ')'       { $$ := $2       }
   ;
%%

procedure main()
   vars := table(0) # initialize all variables to zero
```

```
   write("iyacc Calculator Demo")

   repeat {
     write("expression:")
     yyparse()
   }
end
```

You can use single quoted characters as tokens without declaring them, so `'+'`, `'-'`, `'*'`, `'/'`, `'^'`, `'('`, `')'` are not declared in the above grammar. The unary minus makes use of the %prec keyword to give it priority over binary minus. Note that these are `yacc` notation for small integers; `'+'` is a shorthand for `ord("+")` here, not an Icon cset. Tokens NUM, NAME, ASSIGNMENT, and NEG are declared.

When the parser matches a rule, it executes the Icon code associated with the rule, if there is any. Actions can appear anywhere on the right−hand side of a rule, but their semantics are intuitive only at the end a rule. Vertical bars actually mark alternative rules, so it makes sense to have a (possibly different) action for each alternative. The action code may refer to the value of the symbols on the right side of the rule via the special variables $1, $2... and they can set the value of the symbol on the left of the colon by assigning to the variable $$. The value of a terminal symbol is the contents of variable `yylval`, which is assigned by the scanner. The value of a non−terminal is the value assigned to the $$ variable in the grammar rule that produced that non−terminal. As was mentioned for the `yylval` variable, in other languages, assigning different kinds of values to $$ for different non−terminals is awkward. In Icon, it is very easy because variables can hold values of any type.

Making It All Work From The Command−Line

How do you combine all these tools correctly to build a program? Here is the sequence of commands you would type for creating a calculator given the ilex and iyacc input files presented earlier. You might normally use a "make" program that manages the dependencies of different files and programs, to avoid typing this all by hand, but "make" is beyond the scope of this chapter.

```
$ iyacc –d calc.y               creates calc.icn & y_tab.icn
$ ilex calc_lex.l               creates calc_lex.icn
$ unicon calc.icn calc_lex.icn  creates the program calc
```

The resulting calculator program might be executed in a session such as the following:

```
$ calc                    runs calc
iyacc Calculator Demo
expression: (3 + 5)*2     enter an expression
16                        prints out the answer
expression: x := 16       enter an expression
16                        prints out the answer
expression: 2 * x         enter an expression
32                        prints out the answer
expression: ^D            control−D stop the program
```

Final Tips For Using ilex and iyacc

Tasks requiring the processing of languages with a computer program can vary greatly in complexity from very simple to extremely complex. As the complexity grows, `ilex` and `iyacc` are two tools that can make the scanning and parsing part of the task very easy to manage. Because each tool makes the grammar of the language they are

processing very explicit in the code, programs written with these tools are often easier to maintain than programs with handwritten scanners and parsers. Each tool can be used independently and they also work well together. Adding a little language to your application can make it a more powerful and general tool. Imagine how much less useful spreadsheets would be if you could not enter formulas in the cells. Those formulas are a little language!

Since Icon already has high−level string processing facilities, it is worth asking how `ilex` and `iyacc` relate to the language, or why they are needed. One answer is that they are standard tools expected by compiler writers. Another answer is that they use special−purpose notations and algorithms to deliver higher performance than is provided by Icon's more general and more flexible facilities, and they do this from simpler specifications. A final answer is that these tools are not mutually exclusive of Icon's other features and you should use whatever combination of these tools and Icon's built−in facilities are appropriate to the task at hand.

Summary

This chapter only scratches the surface of all the features provided by `ilex` and `iyacc`. The goal is to introduce you to the capabilities of these tools that are now easy to use with Icon. In fact, in many ways it is much easier to use these tools with Icon than it is to use them with C, thanks to Icon's string processing facilities and built−in data structures.

Part IV: Appendixes

Appendix A: Language Reference

Icon 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 (?).

Immutable Types

Icon'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. Radix integer literals use the format *radix*R*digits*, where *radix* is a base in the range 2 through 36, and *digits* 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.

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 *number*E*integer*.

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 3−1.

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.

Mutable Types

Mutable types have values that may be altered. Changes to a mutable value affect its allocated memory or its associated persistent storage. Mutable types include lists, tables, sets, records, objects, and files, including windows, network connections and databases. These types are described in this Appendix in the entries for constructors that create mutable values. Structure types hold collections of elements that may be of arbitrary, mixed type.

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 that may be accessed via named fields.

Object

Objects are ordered, fixed length sequences of elements that may be accessed via named fields and methods. Field access to object state is a widely deprecated practice.

File

Files are system interface values that correspond to data on secondary storage, areas on users' displays, network connections, or databases. Operations on files are input or output functions that cause side effects on the system outside of the program execution.

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 three kinds of variables: global, local, and static. Variables are declared by introducing one of the three 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.

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 and preinitializes it to the procedure value that corresponds to the code for that procedure.

Local

Local variables are visible only within a single procedure or method, and exist at any particular location 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 one of its ancestor invocations. Variables that are undeclared in any scope are implicitly local, but this dangerous practice is deprecated and 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. A set of class variables is created for each instance (object) of the class. The lifespan of class variables is the same as 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.

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.

&allocated : integer* report memory use

&allocated generates the cumulative number of bytes allocated in heap, static, string, and block regions during the entire program execution.

&ascii : cset ASCII character set

&ascii produces a cset corresponding to the ASCII characters.

&clock : string time of day

&clock produces a string consisting of the current time of day in hh:mm:ss format. See also keyword &now.

&collections : integer* garbage collection activity

&collections generates the number of times memory has been reclaimed in heap, static, string, and block regions.

&column : integer source code column

&column returns the source code column number of the current execution point. This is especially useful for execution monitoring.

&cset : cset universal character set

&cset produces a cset constant corresponding to the universal set of all characters.

&current :co−expression current co−expression

&current produces the co−expression that is currently executing.

&date : string today’s date

&date produces the current date in yyyy/mm/dd format.

&dateline : string time stamp

&dateline produces a human−readable timestamp that includes the day of the week, the date, and the current time, down to the minute.

&digits : cset digit characters

&digits produces a cset constant corresponding to the set of digit characters 0−9.

&dump := integer termination dump

&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 : real — natural log e

&e is the base of the natural logarithms, 2.7182818?

&error := integer — fail on error

&error 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 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 &syserr.

&errornumber : integer — runtime error code

&errornumber is the error number of the last runtime error that was converted to failure.

&errortext : string — runtime error message

&errortext is the error message of the last error that was converted to failure.

&errorvalue : any — offending value

&errorvalue is the erroneous value of the last error that was converted to failure.

&errout : file — standard error file

&errout is the standard error file. It is the default destination to which runtime errors and program termination messages are written.

&eventcode := integer — program execution event

&eventcode indicates the kind of behavior that occurred in a monitored program at the time of the most recent call to EvGet(). This keyword is only supported under interpreters built with execution monitoring support.

&eventsource := co−expression — source of program execution events

&eventsource 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 &source.

&eventvalue := any — program execution value

&eventvalue is a value from the monitored program that was being processed at the time of the last program event returned by EvGet(). This keyword is only supported under interpreters built with execution monitoring support.

&fail : none — expression failure

&fail never produces a result. Evaluating it always fails.

&features : string* platform features

&features generates strings that indicate the nonportable features supported on the current platform.

&file : string current source file

&file is the name of the source file for the current execution point. This is especially useful for execution monitoring.

&host : string host machine name

&host is a string that identifies the host computer Icon is running on.

&input : file standard input file

&input is a standard input file. It is the default source for file input functions.

&lcase : cset lowercase letters

&lcase is a cset consisting of the lowercase letters from a to z.

&letters : cset letters

&letters is a cset consisting of the upper and lowercase letters A−Z and a−z.

&level : integer call depth

&level gives the nesting level of the currently active procedure call. This keyword is not supported under the optimizing compiler, iconc.

&line : integer current source line number

&line is the line number in the source code that is currently executing.

&main : co−expression main task

&main is the co−expression in which program execution began.

&now : integer current time

&now produces the current time as the number of seconds since the epoch beginning 00:00:00 GMT, January 1, 1970. See also &clock

&null : null null value

&null produces the null value.

&output : file standard output file

&output is the standard output file. It is the default destination for file output.

&phi : real — golden ratio

&phi is the golden ratio, 1.61803?

&pi : real — pi

&pi is the value of pi, 3.14159?

&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.

&progname := string — program name

&progname 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 ?

&regions : integer* — region sizes

&regions produces the sizes of the static region, the string region, and the block region. The first result is always zero and 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 with a runtime error. 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 indicates the number of nesting levels to which the program execution should be traced. 0 means no tracing. A negative value turns on tracing to an infinite depth. &trace is set outside the program using the TRACE environment variable or the −t compiler option.

&ucase : cset — upper case letters

&ucase is a cset consisting of all the upper case letters from A to Z.

&version : string — Icon version

&version is a string that indicates which version of Icon is executing.

Graphics Keywords

Graphics keywords are described in [Griswold98]. They are listed here so that you will know to consult that book when reading Icon programs that use graphics.

&col := integer	column location of pointer
&control : null	control key state
&interval : integer	time between input events
&ldrag : integer	left button drag
&lpress : integer	left button press
&lrelease : integer	left button release
&mdrag : integer	middle button drag
&meta : null	meta key state
&mpress : integer	middle button press
&mrelease : integer	middle button release
&rdrag : integer	right button drag
&resize : integer	window resize
&row := integer	row location of pointer
&rpress : integer	right button press
&rrelease : integer	right button release
&shift : null	shift key state
&window := window	standard window
&x := integer	pointer horizontal position
&y := integer	pointer vertical position

Control Structures and Reserved Words

Icon 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.

break *expr* exit loop

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* step increment

The by reserved word supplies a step increment to a to−expression (the default is 1).

case *expr* of { ? } select expression

The case expression selects one of several branches of code to be executed.

class name [: superclass]* (fields) methods [initially] end class declaration

The class declaration introduces a new object type into the program. The class declaration may include a list of superclasses, fields, methods, and an optional initially section.

create *expr* create co−expression

The create expression produces a new co−expression to evaluate *expr*.

default : *expr* default case branch

The default branch of a case expression is taken if no other case branch is taken.

do *expr* iteration expression

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 else branch

The else expression is executed if expr1 fails to produce a result.

end end of declared body

The reserved word end signifies the end of a procedure, method, or class body.

every *expr1* [do *expr2*] generate all results

The every expression always fails, causing expr1 to be resumed for all its results.

fail produce no results

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

global *var* [, *var*]* declare global variables

Reserved word global introduces one or more global variables.

if *expr* then *expr2* [else *expr3*] conditional expression

The if expression evaluates expr2 only if expr1 produces a result.

import *name* [, *name*]* import package

The import declaration introduces the names from package *name* so that they may be used without prefixing them with the package name.

initial *expr* execute on first invocation

The initial expression is executed the first time a procedure or method is invoked.

initially [(parameters)] initialize object

The initially section defines a special method that is invoked automatically when an object is created. If the initially section has declared parameters, they are used as the parameters of the constructor for objects of that class.

invocable *procedure* [, *procedure*]* allow string invocation
invocable all allow string invocation

The invocable declaration indicates that procedures may be used in string invocation.

link *filename* [, *filename*]* link code module

The link declaration indicates that the code in *filename* will be included in the executable when this program is linked. *filename* may be an identifier or a string literal containing a path.

local *var* [, *var*]* declare local variables

The local declaration introduces one or more local variables into the current procedure or method body. Variable declarations must be at the beginning of a procedure or method.

method *name* (*params*) *body* end declare method

The method declaration introduces a procedure that is invoked with respect to instances of an enclosing class declaration. The *params* and *body* are as in procedure declarations, described below.

next *expr* iterate loop

The next expression causes a loop to immediate skip to its next iteration.

not *expr* — negate expression failure

The not expression fails if expr succeeds, and succeeds (producing null) if expr fails.

case *expr* **of** { ? } — introduce case branches

The of reserved word precedes a special compound expression consisting if a sequence of case branches of the form *expr* : *expr*. Case branches are evaluated in sequence until one is found that matches the expression given between the word case and the of.

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.

procedure *name* (*params*) *body* end — declare procedure

The procedure declaration introduces a procedure with the specified parameters and code body. The parameters are a comma−separated list of zero or more variable names. The last parameter may optionally be suffixed by [] indicating it and any 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.

record *name* (*fields*) — declare record

The record declaration introduces a new record type into the program.

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.

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.

static *var* [, *var*]* — declare static variables

The static declaration introduces one or more local variables into the current procedure or method body. Variable declarations must be at the beginning of a procedure or method.

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.

---

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*.

---

expr1 **to** *expr2* generate arithmetic sequence

The to expression produces the integer sequence from *expr1* to *expr2*.

---

until *expr1* [do *expr2*] loop until success

The until expression loops as long as *expr1* fails.

---

while *expr1* [do *expr2*] loop until failure

The while expression loops as long as *expr1* succeeds.

Operators and Built−in Functions

Icon’s built−ins operators and functions utilize automatic type conversion to provide flexibility and ease of programming. 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.

Many 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.

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

! *x* : any* generate elements

The generate operator produces the elements of x. It is a generator. The results are variables that may be assigned. List, record, string, and file elements are generated in order, with string elements consisting of one−letter substrings, and file elements consisting of text lines. !i is equivalent to (1 to i) for integer i.

/ x null test
\ x nonnull test

The null and nonnull tests succeed and produce their operand if it satisfies the test.

− number negate
+ number numeric identity

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 tab/match

The tab/match operator is equivalent to calling tab(match(s)) on its operand.

* x : integer size

The size operator returns the number of elements in structure x.

. x : x dereference

The dereference operator returns the value x.

? x : any random element

The random operator produces a random element from structure x. The result is a variable that may be assigned. If x is a string, ?x produces a random one−letter substring.

| x : x* repeated alternation

The repeated alternation operator generates results from evaluating its operand over and over again in an infinite loop.

~ cset cset complement

The complement operator produces a cset consisting of all characters not in its operand..

^ co−expression refresh 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

number1 ^ *number2*	power
number1 * *number2*	multiply
number1 / number2	divide
number1 % number2	modulo
number1 + number2	add
number1 − number2	subtract

The arithmetic operators may be augmented.

set1 ** set2	intersection
set1 ++ set2	union
set1 −− set2	difference

The set operators work on sets or csets. They may be augmented.

x . name	field
object . name (params)	method invocation
object $ superclass .name (params)	superclass method invocation

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.

number1 = number2	equal
number1 ~= number2	not equal
number1 < number2	less than
number1 <= number2	less or equal
number1 > number2	greater than
number1 >= number2	greater or equal
string1 == string2	string equal
string1 ~== string2	string not equal
string1 << string2	string less than
string1 <<= string2	string less or equal
string1 >> string2	string greater than
string1 >>= string2	string greater or equal
x1 === x2	equivalence
x1 ~=== x2	non equivalence

Relational operators produce their right operand if they succeed. They may be augmented.

var := expr	assign
var1 :=: var2	swap
var <− expr	reversible assignment
var1 <−> var2	reversible swap

The several assignment operators all require variables for their left operands, and swap operators also require variables for their right operands.

string ? expr — scan string

The string scanning operator evaluates *expr* with &subject equal to string and &pos starting at 1. It may be augmented.

[x] @ co−expression — 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.

string1 || string2 — concatenation
list1 ||| list2 — list 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.

x1 & x2 — conjunction
expr1 | expr2 — 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.

x1 \ integer — limitation

The limitation operator fails if it is resumed after its left operand has produced a number of results equal to its right operand.

(expr [, expr]*) — mutual evaluation
p (expr [, expr]*) — invocation

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.

[ ] — empty list creation
[expr [, expr]* **]** — list creation
expr1 **[** expr2 [, expr]* **]** — subscript
expr1 **[** expr2 : expr3 **]** — subsection
expr1 **[** expr2 +: expr3 **]** — forward relative subsection
expr1 **[** expr2 −: expr3 **]** — backward relative subsection

Square brackets are used to create and initialize lists. 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.

Subscripting selects an element from a structure and allows that element to be assigned or for its value to be used. Subsectioning only works on strings and lists. For strings, the subsection is a variable if the string was a variable, and assignment to the subsection

will modify the original variable. For lists, a subsection is a new list that contains a copy of the elements from the original list.

---

expr1 ; expr2 — bound expression

A semicolon bounds expr1. Once expr2 is entered expr1 cannot be resumed for more results. The result of 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.

{ expr [; expr]* } — compound expression
p { expr [; expr]* } — programmer defined control structure

Curly brackets allow a sequence of bounded expressions to be treated as a single expression.

Built−in Functions

Icon's built−in functions are a key element of its ease of learning and use. They provide substantial functionality in a consistent and memorizable manner.

In addition to automatic type conversion, Icon's built−in functions make extensive use of optional parameters with default values. Default values are indicated in the function descriptions, 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(number) : number — absolute value

abs(N) produces the maximum of N or −N.

---

acos(real) : real — arc cosine

acos(r1) produces the arc cosine of r1. The argument is given in radians.

---

any(cset, string, integer, integer) : 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(procedure) : integer — number of arguments

args(p) produces the number of arguments expected by 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.

---

asin(real) : real — arc sine

asin(r1) produces the arc sine of r1. The argument is given in radians.

atan(real, real: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.

bal(cset:&cset, cset:'(', cset:')', string, integer, integer) : integer* — balance string

String scanning function bal(c1,c2,c3,s,i1,i2) generates the integer positions in s at which a member of c1 in s[i1:i2] is balanced with respect to characters in c2 and c3.

center(string, integer:1, string:" ") : string — center string

center(s1,i,s2) produces a string of i characters. If i > *s1 then s1 is padded equally on the left and right with s2 to length i. If i < *s1 then the center i characters of s1 are produced.

char(integer) : string — encode character

char(i) produces a string consisting of the character encoded by integer i.

chmod(f, m) : null? — file permissions

chmod(f, m) sets the access permissions ("mode") of a string filename (or on UNIX systems, an open file) f to a string or integer mode m. The mode indicates the change to be performed. The string is of the form

[ugoa]*[+−=][rwxRWXstugo]*

The first group describes the set of mode bits to be changed: u is the owner set, g is the group and o is the set of all others. The character a designates all the fields. The operator (+−=) describes the operation to be performed: + adds a permission, − removes a permission, and = sets a permission. The permissions themselves are:

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

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 described here; the mode bits that are used is a property of the filesystem on which the file resides.

close(file) : file | integer — close file

close(f) closes file, pipe, window, or database f and returns any resources associated with it to the operating system. If f was a pipe or socket, close() returns the integer exit status of the connection, otherwise it returns the closed file.

cofail(co−expression) : any — transmit co−expression failure

cofail(CE) causes the current activation of co−expression CE to fail.

collect(integer:0, integer:0) : null — collect garbage

collect(i1,i2) calls the garbage collector to ensure i2 bytes in region i1. i1 can be 0 (no region in particular) 1 (static region) 2 (string region) or 3 (block region).

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(real) : real — cosine

cos(r1) produces the cosine of r1. 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(integer) : string — format a time value into local time

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(database) : list — get column information

dbcolumns(db) produces a list of record (catalog, schema, tablename, colname, datatype, typename, colsize, buflen, decdigits, numprecradix, nullable, remarks) entries. Datatype and typename are SQL−dependent and data source dependent, respectively. Colsize gives the maximum length in characters for SQL_CHAR or SQL_VARCHAR columns.. Decdigits gives the number of significant digits right of the decimal. Numprecradix specifies whether colsize and decdigits are specified in bits or decimal digits.. 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.

dbdelete(database, string[]) : integer — delete database rows

dbdelete(db, filter...) deletes rows from db that satisfy filters. The corresponding SQL statement looks like DELETE db WHERE filter. Warning: if the filter criteria are omitted, the database will be emptied by this operation!

dbdriver(database) : record — database driver information

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. Ver and odbcver are the driver and ODBC version numbers. Name and dsn are the driver filename and Windows Data Source Name associated with the connection.

dbfetch(database) : record? — fetch row from query result

dbfetch(db) produces the next row from the current query, in a record whose field names and types are determined by the columns specified in the current query. If 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 dbfetch() repeatedly until it fails.

dbinsert(database, record) : integer — insert database row

dbinsert(db, row) inserts a record as a tuple into db.

dbkeys(database) : list — database key information

dbkeys(db) produces a list of record (col, seq) pairs containing information about the primary keys in the table associated with db. The column name and sequence number for each key are given.

dblimits(database) : record — database operation limits

dblimits(db) produces a record (maxbinlitlen, maxcharlitlen, maxcolnamelen, maxgroupbycols, maxorderbycols, maxindexcols, maxselectcols, maxtblcols, maxcursnamelen, maxindexsize, maxrownamelen, maxprocnamelen, maxqualnamelen, maxrowsize, maxrowsizelong, maxstmtlen, maxtblnamelen, maxselecttbls, maxusernamelen) that contains the upper bounds of the database for many parameters.

dbproduct(database) : record — database name

dbproduct(db) produces a record (name, ver) that gives the name and the version of the DBMS product containing db.

dbselect(database, string,string,string) : integer — database selection query

dbselect(db, columns, condition, ordering) selects columns from db. Columns defaults to all, condition defaults to unconditionally, and ordering defaults to unordered. The corresponding SQL statement would be SELECT columns FROM db WHERE condition ORDER BY ordering.

dbsql(database, string) : integer — execute SQL statement

dbsql(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. dbsql() can leave the database in an arbitrary state and should be used with care. For example, if a call to dbsql() directs the

server elsewhere, db may no longer be associated with the table on which it was opened, causing subsequent operations to fail.

---

dbtables(database) : list — get ODBC column information

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.

---

dbupdate(database, record) : integer — update database row

dbupdate(db, row) updates the database tuple corresponding to row.

---

delay(integer) : null — delay for i milliseconds

delay(i) pauses the program for at least i milliseconds.

---

delete(x1, x2) : x1 — delete element

delete(x1, x2) deletes element x2 from set, table, or list x1 if it is there. In any case, it returns x1. If x1 is a table or set, x2 is a key of arbitrary type. If x1 is a list, x2 is an integer index of the element to be deleted.

---

detab(string, integer:9,...) : string — replace tabs

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.

---

display(integer:&level, file:&errout, co−expression:&current) : null — write variables

display(i,f) writes the local variables of i most recent procedure activations, plus global variables, to file f.

---

dtor(real) : real — convert degrees to radians

dtor(r1) produces the equivalent of r1 degrees, expressed in radians.

---

entab(string, integer:9,...) : string — replace spaces

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.

---

errorclear() : null — clear error condition

errorclear() resets keywords &errornumber, &errortext, and &errorvalue to indicate that no error is present.

---

event(integer, any, co−expression) : any — transmit event

event(x, y, C) transmits an event with event code x and event value y to a monitoring co−expression C.

eventmask(co−expression, cset) : cset | null — get/set event mask

eventmask(ce) returns the event mask associated with the program that created ce, or &null if there is no eventmask. eventmask(ce,cs) sets that program's event mask to cs.

exit(integer:*normalexit*) — exit process

exit(i) terminates the current program execution, returning status code i. The default is the platform−dependent exit code that indicates normal termination (0 on most systems).

exp(real) : real — exponential

exp(r1) produces the result of &e ^ r1.

fetch(database, string) : string | row — fetch database value

fetch(d, k) fetches the value corresponding to key k from a DBM or SQL database d. The result is a string (for DBM databases) or a row (for SQL databases). If the string k is omitted, fetch(d) produces the next row in the current selection, for SQL database d and advances the cursor to the next row.

fieldnames(record) : string* — get field names

fieldnames(r) produces the names of the fields in record r.

find(string, string, integer, integer) : 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(file, string) : null? — apply or remove file lock

An advisory lock is applied 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(file) : 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(list) : any? — get element from queue

get(L) returns an element which is removed from the head of the queue L.

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(string) : 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(co−expression) : string* — name the global variables

globalnames(ce) generates the names of the global variables in the program that created co−expression ce.

gtime(integer) : 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(integer, integer) : integer — bitwise and

iand(i1, i2) produces the bitwise and of i1 and i2.

icom(integer) : integer — bitwise complement

icom(i) produces the bitwise complement (one’s complement) of i.

image(any) : integer — 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, list or 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(integer, integer) : integer bitwise or

ior(i1, i2) produces the bitwise or of i1 and i2.

ishift(integer, integer) : 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.

ixor(integer, integer) : integer bitwise xor

ixor(i1, i2) produces the bitwise exclusive or of i1 and i2.

kbhit() : null? 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 successive keys (entry values) from table T. key(L) generates successive indices from 1 to *L in list L.

left(string, integer:1, string:" ") : 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(string, list, file:&input, file:&output, file:&errout, int, int, int) : co−expression
load Icon 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(string, string) : 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(co−expression, integer) : string* local variable names

localnames(ce,i) generates the names of local variables in co−expression ce, i levels up from the curent procedure invocation.

log(real, real:&e) : real logarithm

log(r1,r2) produces the logarithm of r1 to base r2.

many(cset, string, integer, integer) : integer? many characters

String scanning function many(c,s,i1,i2) produces the position in s after the longest initial sequence of members of c within s[i1:i2].

map(string, string:&ucase, string:&lcase) : string map string

map(s1,s2,s3) maps s1, using s2 and s3. The resulting string will be a copy of s1, with the exception that any of s1's characters that appear in s2 are replaced by characters at the same position in s3.

match(string, string, integer, integer) : integer match string

String scanning function match(s1,s2,i1,i2) produces i1+*s1 if s1==s2[i1+:*s1], but fails otherwise.

member(x1, x2) : x1? test membership

member(x1, x2) returns x1 if x2 is a member of set or table x1 but fails otherwise.

mkdir(string, mode) : null? create directory

mkdir(path, mode) creates a new directory named path with mode mode. The mode can be numeric or a string of the form accepted by chmod().

move(integer:1) : string move 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.

name(v, co−expression:&current) : string variable name

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 convert to 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(string, string:"rt", ...) : file? — open file

open(s1, s2, ...) opens a file named s1 with mode s2 and attributes given in trailing arguments. The modes recognized by open() are:

"r"	open the file for reading
"w"	open the file for writing
"a"	open the file for appending
"b"	open the file for both reading and writing (b does not mean binary!)
"p"	execute a program given by command line s1 and open a pipe to it
"c"	create a new file and open it
"t"	open the file in text mode, with newlines translated
"u"	open the file in a binary untranslated mode
"na"	listen on a TCP network socket
"n"	connect to a TCP network socket
"nau"	listen on a UDP network socket
"nu"	connect to a UDP network socket
"d"	open a DBM database
"q"	open a connection to a SQL database

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 s1 is the name of the socket to connect to: if of the form "s:i", it is an Internet domain socket on host s and port i; otherwise, it's the name of a Unix domain socket. If the host name is null, it represents the current host.

For a UDP socket, 'connect' means that 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, read or reads cannot be performed on a UDP socket; use receive. UDP sockets must be in the INET domain, i.e. the address must have a colon.

For a DBM database, only one modifier character may be used: if s1 is "dr" it indicates that the database should be opened in read−only mode.

---

opmask(co−expression, cset) : 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.

---

ord(string) : integer — ordinal value

ord(s) produces the integer ordinal (value) of s, which must be of size 1.

---

paramnames(co−expression, integer: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(co−expression) : co−expression — parent program

parent(ce) given a ce, return &main for that ce's parent. This is interesting only when programs are dynamically loaded using the load() function.

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(list) : any? — pop from stack

pop(L) removes an element from the top of the stack (L[1]) and returns it.

pos(integer) : integer? — test scanning position

pos(i) tests whether &pos is at position i in &subject.

proc(any, integer:1) : procedure? — convert to procedure

proc(x,i) converts x to a procedure if that is possible. Parameter i is used to resolve ambiguous string names. i 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.

pull(list) : any? — remove from list end

pull(L) removes and produces an element from the end of list L, if L is nonempty.

push(list, 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 same list that is passed as its first parameter, with the new elements added.

put(list, x1, ..., xN) : list — add to list end

put(L, x1, ..., xN) puts elements onto the end of list L.

query(database, string) : null? — query database

query(d, s) sends a SQL query to database d. A new set of rows is selected, and the cursor is reset to the beginning of that set of rows. Rows are subsequently obtained from the database using fetch().

read(file:&input) : string? — read line

read(f) reads a line from file f. The end of line marker is discarded.

reads(file:&input, integer:1) : string? read characters

reads(f,i) reads 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.

real(any) : real? convert to real

real(x) converts x to a real, or fails if the conversion cannot be performed.

receive(file) : 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(string) : null? remove file

remove(s) removes the file named s.

rename(string, string) : null? rename file

rename(s1,s2) renames the file named s1 to have the name s2.

repl(x, integer) : 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(string, integer:1, string:" ") : 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(string) : null? remove directory

rmdir(d) removes directory d. rmdir() fails if directory d is not empty or does not exist.

rtod(real) : real convert radians to degrees

rtod(r1) produces the equivalent of r1 radians, expressed in degrees.

runerr(integer, any) runtime error

runerr(i,x) produces runtime error i with value x. Program execution is terminated.

seek(file, 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 priimary key.

---

select(x1, x2, ?) : list — files with available input

select(files?, timeout) waits for a input to become available on any of several files, including 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 dbm files cannot be arguments to select().

---

send(string, string) : null — send datagram

send(s1, s2) sends a UDP datagram to the address s1 (in host:port format) with the contents s2.

---

seq(integer:1, integer: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(list:[]) : set — create set

set(L) creates a set with members in list L. The members are linked into hash chains which are arranged in increasing order by hash number.

---

sort(x, integer:1) : list — sort structure

sort(x, i) sort 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, integer:1) : list — sort by field

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.

---

sqrt(real) : real — square root

sqrt(r) produces the square root of r.

stat(f) : record — get file information

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 mtime() functions. The mode is a string similar to the long listing option of the UNIX ls(1) command. For example, "−rwxrwsr−x" represents a plain file with a mode of 2775 (octal).

staticnames(co−expression:&current, integer:0) : string* — static variable names

staticnames(ce,i) generates the names of static variables in the procedure i levels above the current activation in ce.

stop(x, ?) : — stop execution

stop(args) halts execution after writing out its string arguments, followed by a newline, to &errout. If any argument is a file, subsequent arguments are written to that file instead of &errout. The program exit status indicates that an error has occurred.

string(x) : string? — convert to string

string(x) converts x to a string and returns the result, or fails if the value cannot be converted.

system(x, file:&input, file:&output, file:&errout, string) : 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. If the waitflag argument is "nowait", system will return immediately after spawning the new process. The default is for it to wait for the spawned process to exit. The return value is the exit status from the process, unless the waitflag was set to "nowait", in which case the return value is the process id of the new process.

tab(integer: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(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.

tan(real) : real — tangent

tan(r) produces the tangent of r in radians.

trap(string, procedure) : 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, iconc!

trim(string, cset:' ') : string — trim string

trim(s,c) removes trailing characters in c from s.

truncate(f, integer) : null? — 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.

type(x) : string — type of value

type(x) returns a string that indicates the type of x.

upto(cset, string, integer, integer) : integer* — find characters in set

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(string, integer, integer) : null — file access/modification times

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 filesystems support only a subset of these times.

variable(string, co−expression:&current, integer:0) : any? — get variable

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.

where(file) : integer? — file position

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(x, ?) : x — write text line

write(args) writes out its string arguments, followed by a newline, to &output. If any argument is a file, subsequent arguments are written to that file instead of &output.

writes(x, ?) : x — write strings

write(args) writes out its string arguments to &output. If any argument is a file, subsequent arguments are written to that file instead of &output.

Graphics Functions

Unlike the other built−in functions, the names of built−in graphics functions begin with upper case. User functions, and library functions described in the next appendix, may begin with either upper or lower case. Built−in graphics functions are listed here so that you do not inadvertently use the names and get global identifiers where locals are intended. These functions are described in [Griswold98]. You should consult that book when reading Icon programs that use graphics.

Active() − produce the next active window

Alert(w,volume) − alert the user

Bg(w,s) − background color

Clip(w, x, y, w, h) − set context clip rectangle

Clone(w, attribs...) − create a new context bound to w’s canvas

Color(argv[]) − return or set color map entries

ColorValue(w,s) − produce RGB components from string color name

CopyArea(w,w2,x,y,width,height,x2,y2) − copy area
Couple(w,w2) − couple canvas to context

DrawArc(argv[]) − draw arc

DrawCircle(argv[]) − draw circle

DrawCurve(argv[]) − draw curve

DrawImage(w,x,y,s) − draw bitmapped figure

DrawLine(argv[]) − draw line
DrawPoint(argv[]) − draw point
DrawPolygon(argv[]) − draw polygon
DrawRectangle(argv[]) − draw rectangle
DrawSegment(argv[]) − draw line segment
DrawString(argv[]) − draw text
EraseArea(w,x,y,width,height) − clear an area of the window
EvGet(c,flag) − read through the next event token having a code matched by cset c.
Event(W, i) : a − return next window event
Fg(w,s) − foreground color

FillArc(arcs?)− fill arc

FillCircle(circles?)− draw filled circle

FillPolygon(points?) − fill polygon

FillRectangle(rectangles?)− draw filled rectangle
Font(w,s) − get/set font

FreeColor(colors?) − free colors

GotoRC(w,r,c) − move cursor to a particular text row and column

GotoXY(w,x,y) − move cursor to a particular pixel location

Lower(w) − lower w to the bottom of the window stack
NewColor(w,s) − allocate an entry in the color map
PaletteChars(w,p) − return the characters forming keys to palette p
PaletteColor(w,p,s) − return color of key s in palette p
PaletteKey(w,p,s) − return key of closest color to s in palette p
Pattern(w,s) − sets the context fill pattern by string name
Pending(w,x[]) − produce a list of events pending on window
Pixel(w,x,y,width,height) − produce the contents of some pixels
QueryPointer(w) − produce mouse position
Raise(w) − raise w to the top of the window stack
ReadImage(w, s, x, y, p) − load image file
TextWidth(w,s) − compute text pixel width
Uncouple(w) − uncouple window
WAttrib(argv[]) − read/write window attributes
WDefault(w,program,option) − get a default value from the environment
WFlush(w) − flush all output to window w
WSync(w) − synchronize with server
WriteImage(w,filename,x,y,width,height) − write an image to a file

One of the graphics functions, Event(w, i), has been extended in Unicon as follows. The new, second argument i is a timeout value in milliseconds. If the timeout expires before an event is available, then Event() fails. The default for i signifies that Event() should wait forever for the next window event.

Preprocessor

Icon 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:

$define *symbol text* symbolic substitution

All subsequent occurrences of *symbol* are replaced by the *text* within the current file. Note that Icon $define does not have arguments, unlike C.

$include *filename* insert source file

The named file is inserted into the compilation in place of the $include line.

$ifdef *symbol*	conditional compilation
$ifndef *symbol*	conditional compilation
$else	conditional alternative
$endif	end of conditional code

The subsequent lines of code, up to an $else or $endif, are discarded unless *symbol* is defined by some $define directive. $ifndef reverses this logic.

$error *text*	compile error

The compiler will emit an error with the supplied text as a message.

$line *number* [*filename*]	source code line
#line *number* [*filename*]	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.

$undef *symbol*	remove symbol definition

Subsequent occurrences of *symbol* are no longer replaced by any substitute text.

$(	**{**
$)	**}**
$<	**[**
$>	**]**

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:

Preprocessor Symbol	**Feature**
_V9	Version 9
_AMIGA	Amiga
_ACORN	Acorn Archimedes
_ATARI	Atari ST
_CMS	CMS
_MACINTOSH	Macintosh
_MSDOS_386	MS−DOS/386
_MS_WINDOWS_NT	MS Windows NT
_MSDOS_286	MS−DOS/286
_MSDOS	MS−DOS
_MVS	MVS
_OS2	OS/2
_PORT	PORT
_UNIX	UNIX
_POSIX	POSIX
_DBM	DBM

_VMS	VMS
_ASCII	ASCII
_EBCDIC	EBCDIC
_CO_EXPRESSIONS	co−expressions
_DYNAMIC_LOADING	dynamic loading
_EVENT_MONITOR	event monitoring
_EXTERNAL_FUNCTIONS	external functions
_KEYBOARD_FUNCTIONS	keyboard functions
_LARGE_INTEGERS	large integers
_MULTITASKING	multiple programs
_PIPES	pipes
_RECORD_IO	record I/O
_SYSTEM_FUNCTION	system function
_NETWORK	network
_GRAPHICS	graphics
_X_WINDOW_SYSTEM	X Windows
_MS_WINDOWS	MS Windows
_WIN32	Win32
_WIN16	Win16
_PRESENTATION_MGR	Presentation Manager
_ARM_FUNCTIONS	Archimedes extensions
_DOS_FUNCTIONS	MS−DOS extensions

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 allows a program to convert runtime errors into expression failure. In the event that 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
143 malformed event queue
144 window system error
145 bad window attribute
146 incorrect number of arguments to drawing function
147 window attribute cannot be read or written as requested
160 cannot open file
161 bad file attribute
162 cannot open socket
170 string or integer expected
171 posix header file not included
172 posix record overridden by global value
173 directory opened for writing
174 directory or database invalid as file
175 invalid mode string
176 invalid signal
177 invalid operation to flock/fcntl
178 invalid procedure type
179 fdup of closed file
180 low−level read or select mixed with buffered read
181 not a network connection
182 not a UDP socket
183 invalid protocol name
184 invalid permission string for umask
190 database expected
201 division by zero
202 remaindering by zero
203 integer overflow
204 real overflow, underflow, or division by zero
205 invalid value

206 negative first argument to real exponentiation
207 invalid field name
208 second and third arguments to map of unequal length
209 invalid second argument to open
210 non−ascending arguments to detab/entab
211 by value equal to zero
212 attempt to read file not open for reading
213 attempt to write file not open for writing
214 input/output error
215 attempt to refresh &main
216 external function not found
301 evaluation stack overflow
302 memory violation
303 inadequate space for evaluation stack
304 inadequate space in qualifier list
305 inadequate space for static allocation
306 inadequate space in string region
307 inadequate space in block region
308 system stack overflow in co−expression
401 co−expressions not implemented
402 program not compiled with debugging option
500 program malfunction
600 vidget usage error

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.

1001 Operation not permitted
1002 No such file or directory
1003 No such process
1004 Interrupted system call
1005 I/O error
1006 No such device or address
1007 Arg list too long
1008 Exec format error
1009 Bad file number
1010 No child processes
1011 Try again
1012 Out of memory
1013 Permission denied

1014 Bad address
1016 Device or resource busy
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 complete Icon Program Library consists of approximately 20 directories of code, data, and supporting documentation. This material is included in its entirety on the accompanying CD−ROM. Documenting the entire library would require many hundreds of pages; furthermore, it is a moving target. New library modules are written all the time. This appendix serves as a reference summary. It presents details for selected modules and programs that are among the most useful, general, and portable components the IPL has to offer. In our opinion they constitute a "Best of the Icon Program Library" collection worthy of greater attention, and we hope this appendix results in more use of the IPL. The initials of each author are included in the module description. A list of authors' names appears at the end of the appendix. We have no doubt overlooked some excellent library modules and programs in this selection process, and for that we apologize. 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)

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 is for programs that process 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 from an address, 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 (REG)

---

ansi

This module implements a subset of the ANSI terminal control sequences. The names of the procedures are taken directly from the ANSI names. If it is necessary to use these routines with non−ANSI devices, link in `iolib` and (optionally) `iscreen`. Not all ANSI terminals support even the limited subset of control sequences included in this file. (REG, RLG)

`CUB(i:integer)` moves the cursor left i columns
`CUD(i:integer)` moves the cursor down i rows
`CUF(i:integer)` moves the cursor right i columns
`CUP(i:integer,j:integer)` moves the cursor to row i, column j
`CUU(i:integer)` moves the cursor up i rows
`ED(i:integer:2)` erases screen: i = 0: to end; i = 1: from beginning; i = 2, all
`EL(i:integer:0)` erases current row: i = 0: to end; i = 1: from beginning; i = 2, all
`SGR(i:integer:0)` sets video attributes: 0: off; 1: bold; 4: underscore; 5: blink; 7: reverse

---

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))`. (REG)

argparse

`argparse(s:string) : list` parses `s` as if it was a command line and puts the components in a list, which is returned. At present, it does not accept any escape conventions. (REG)

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 the specified lower bounds, upper bounds, and with each array element having the specified initial value.

`ref_array(A, i1, i2, ...)` references the i1−th i2−th ... element of A. (REG)

asciinam

`asciiname(s:string):string` returns the mnemonic name of the single unprintable ASCII character `s`. (RJA)

base64

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. (DAG)

basename

`basename(name, suffix) : string` strips off any path information and removes the specified suffix, if present. If no suffix is provided, the portion of the name up to the first "." is returned. It should work under at least UNIX, MS−DOS, and the Macintosh. (REG, CAS)

binary

This collection of procedures supports 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, value`$_i$`, ...) : packed_binary_string` packs the *value* arguments into a binary structure, returning the 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 is a sequence of characters that give the order and type of values, as follows:

a	ASCII string, null padded (unstripped for unpack()).
A	ASCII string, space padded (trailing nulls and spaces are stripped for unpack()).
b	bit string, low−to−high order.
B	bit string, high−to−low order.
h	hexadecimal string, low−nibble−first.
H	hexadecimal string, high−nibble−first.
c	signed char value.
C	unsigned char value.
s	signed short value.
S	unsigned short value.
i	signed integer value.
I	unsigned integer value.
l	signed long value.
L	unsigned long value.
n	short in "network" order (big−endian).
N	long in "network" order (big−endian).
v	short in VAX order (little endian).
V	long in VAX order (little endian).
f	single−precision float in IEEE Motorola format.
d	double precision floats in IEEE Motorola format.
e	extended−precision float in IEEE Motorola format 80−bit.
E	extended−precision float in IEEE Motorola format 96−bit.
x	Skip forward a byte (null−fill for pack()).
X	Back up a byte.
@	Go to absolute position (null−fill if necessary for pack()).
u	A uuencoded/uudecoded string.

Each letter may optionally be followed by a number that gives a count. Together the letter and the count make a field specifier. Letters and numbers can be separated by white space that will be ignored. Types A, a, B, b, H, and h 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()`. (RJA)

---

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`. (RJA)

---

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`. (REG)

---

bitstr, bitstrm

These two modules operate on numeric values represented by strings of an arbitrary number 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.

`BitStringPut(bitString, nbits, value:0)` is called with three arguments to put bits into a `BitString`, and called with a single argument to obtain the resulting string value. An example illustrates how `BitStringPut()` is used:

```
record bit_value(value, nbits)
...
bitString := BitString("")
while value := get_new_value() do # loop to append to string
   BitStringPut(bitString, value.nbits, value.value)
resultString := BitStringPut(bitString) # output any buffered
bits
```

`BitStringPut(bitString)`, as well as producing the complete string, pads the buffered string to an even character boundary. This can be done during construction of a bit string if the effect is desired. The "value" argument defaults to zero.

`BitStringGet(bitString, n:integer):integer` extracts n bits from a BitString and returns them as an integer. An example is the best way to show how `BitStringGet()` is used:

```
record bit_value(value, nbits)
...
bitString := BitString(string_of_bits)
while value := BitStringGet(bitString, nbits) do
   # do something with value
```

`BitStringGet()` fails when too few bits remain to satisfy a request. However, if bits remain in the string, subsequent calls with fewer bits requested may succeed. A negative *nbits* value gets the value of the entire remainder of the string, to the byte boundary at its end.

Module `bitstrm` provides procedures for reading and writing integer values made up of an arbitrary number of bits, stored without regard to character boundaries. An example is the best way to show how `BitStreamWrite()` is used:

```
record bit_value(value, nbits)
...
BitStreamWrite() #initialize
while value := get_new_value() do # loop to output values
BitStreamWrite(outfile, value.nbits, value.value)
```

```
BitStreamWrite(outfile) # output any buffered bits
```

`BitStreamWrite(outproc)` outputs the complete string and pads the output to an even character boundary. This can be done during construction of a bit string if the effect is desired. The *value* argument defaults to zero. `BitStreamRead()` is illustrated by the following example:

```
BitStreamRead()
while value := BitStreamRead(infile, nbits) do
   # do something with value
```

`BitStringRead()` fails when too few bits remain to satisfy a request. (RJA)

---

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:

```
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...
   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. (CAS)

---

calls

These procedures deal with procedure invocations that are encapsulated in records.

`record call(proc, args)` encapsulates a procedure to be called and its argument list.
`invoke(call) : any*` invokes a procedure with an argument from a call record.
`call_image(call) : string` produces a string image of a call.
`make_call(string) : call` makes a call record from a string that looks like an invocation.
`make_args(string) : list` makes an argument list from a comma−separated string.

`call_code(string) : 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`. (REG)

---

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(?)`, `uncaptured_writes(?)` and `uncaptured_stop(?)` 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. (DAG)

---

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. (NJL)

`anycl(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.

`balcl(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.

`findcl(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.

`manycl(c, s, i1, i2)` succeeds and 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))`.

matchcl(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. (JvM, CLJ)

`global cgi : table` is a library global variable whose keys are the names of input fields in the invoking HTML page's form, and whose values are whatever the user typed in those input fields.

`cgiInput(type, name, values)` writes HTML INPUT tags of a particular type with a certain name for each element of a list of values. The first value's input tag is CHECKED.

`cgiSelect(name, values)` writes an HTML SELECT tag with a certain name and embedded OPTION tags for each element of a list of values. The first value's OPTION tag is SELECTED.

`cgiXYCoord(hlst) : string` is used with an ISMAP to check the x and y coordinates. If they 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(table)` prints the keys and values in a 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 a hex char from 0 to F.

`cgiHexChar(c1,c2)` produces an 8−bit char value corresponding to two hex digits encoded as chars.

`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)` selectively copies out parts of a named (HTML) file, writing out either anything between ALL and the value that are passed into the procedure.

`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 include 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.

`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 kinds of values, including structures of arbitrary complexity and even loops. For scalar types −− null, integer, real, cset, and string −− `decode(encode(x)) === x`. For structure types −− list, set, table, and record types −− `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 all characters in a way that is safe to write to a file and read it back.

Links: `escape`, `gener`, `procname`, `typecode`. Requires: co−expressions. See also: `object.icn`. (REG)

---

colmize

`colmize(L:list,mx:80,sp:2,mi:0,tag,tagsp:2,tagmi:0,roww,dist) : string*` arranges a number of 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 in column−wise order unless roww is nonnull; by default the sequence runs down the first column, then down the second, etc. `colmize()` goes to great lengths to print the items in as few vertical lines as possible. (RJA)

---

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. (RLG)

Example:

```
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. (REG)

`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 number `x2` from complex number `x1`
`cpxstr(x) : complex` convert complex number `x` to string representation
`strcpx(s) : complex` convert string representation of complex number to complex number

conffile

This module supports the parsing of configuration files into Icon structures for easy access. The service is similar to standard 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 in to 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 defaults
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.

```
cfg := ReadDirectives([".mycfg", "my.cfg", &input], tos)
WOpen("label=" || ( \ (cfg["mylabel"]) | "untitled"))
```

Links: lastc. (DAG)

---

convert

This module contains numeric conversions between bases. There are several other procedures related to conversion that are not yet part of this module. (REG)

`exbase10(i, j) : string` converts base−10 integer i to base j.
`inbase10(s, i) : integer` converts base−i integer s to base 10.
`radcon(s, i, j) : integer` converts base−i integer s to base j.

---

created

`created(string) : integer` returns the number of structures of a given type that have been created. It artificially increments the count, and allocates garbage. Links: `serial`. (REG)

---

currency

`currency(amt, wid:0, neg:"−", frac:2, whole:1, sign:"$", decimal:".", comma:",") : string` formats `amt` in a currency format that defaults to U.S. currency. `amt` can be a real, integer, or numeric string. `wid` is the output field width, in which the amount is right adjusted. The returned string will be longer than `width` if necessary to preserve significance. `neg` is the character string to be used for negative amounts, and is placed to the right of the amount. `frac` and `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 `sign` prefixes the returned string. The characters used for decimal point and comma may also be supplied. (RJA)

---

datecomp

These procedures do simple date comparisons. The parameters are strings of the form mm/dd/yyyy or `&date`−compatible yyyy/mm/dd. (CSM)

dgt(date1:string, date2:string) : null? succeeds if date1 is later than date2.
dlt(date1:string, date2:string) : null? succeeds if date1 is earlier than date2.
deq(date1:string, date2:string) : null? succeeds if date1 is equal to date2.
futuredate(date:string) : null? succeeds if date is in the future.
pastdate(date:string) : null? succeeds if date is in the past.
getmonth(date:string) : string returns the month portion of a date.

getday(date:string) : string returns the day portion of a date.
getyear(date:string) : string returns the year portion of a date.

datefns

These date and calendar procedures represent dates using the record type defined below. They are adaptations of C functions from "The C Programming Language" (Kernighan and Ritchie, Prentice−Hall) and "Numerical Recipes in C" (Press et al, Cambridge). (CH)

`record date_rec(year, month, day, yearday, monthname, dayname)`
`initdate()` initializes the global data before using the other functions.
`today() : date_rec` produces a computationally useful value for today's date
`julian(date) : integer` converts a date_rec to a Julian day number
`unjulian(julianday) : date_rec` 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 line to `&output` with a basic date string preceded by a leadin

datetime

These procedures provide miscellaneous date and time operations .See also: `datefns.icn`. (RJA, REG)

`global DateBaseYear : 1970` is a time origin for several functions. If an environment variable by the same name is present, its value is used instead of 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 produces a date in the same format as Icon's &dateline.

`SecToUnixDate(sec,hoursFromGmt) : string` returns a date and time in typical UNIX format: Jan 14 10:24 1991.

`calendat(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.

`dayoweek(day, month, year) : string` produces the day of the week for the 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)` returns the Julian Day Number for the specified 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.

---

dif

`dif(strm:list,compare:"===",eof,group:groupfactor):list*` generates differences between input streams. Results are returned as 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 declared as: `record diff_rec(pos,diffs)`. `dif()` fails if there are no differences.

`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:

Type	Action
file	file is "read" to get records
co−expression	co−expression is activated to get records
list	records are "gotten" (`get()`) from the list
diff_proc	a record type defined in `dif` to allow procedures supplied by `dif`'s caller to be called to get records. Diff_proc has two fields, the procedure to call and the argument to pass to it. Its definition looks like this: `record diff_proc(proc,arg)`

`compare` is a procedure that succeeds if two records are "equal", 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.

`eof` is an object that is distinguishable from other objects in the stream.

`group` 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. (RJA)

---

digitcnt

`digitcnt(file:&input) : list` counts the number of each digit in a file and returns a ten−element list with the counts. (REG)

---

ebcdic

These procedures assist in use of the ASCII and EBCDIC character sets, regardless of the native character set of the host. For example, `Ascii128()` returns a 128−byte string of ASCII characters in numerical order, replacing `&ascii` for applications which might run on an EBCDIC host. (AB)

---

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 not quite right 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. (REG)

---

escapesq

These procedures manipulate escape sequences in Icon (or C) character string representations. (RJA)

`escapeseq() : string` is a matching procedure for Icon string escape sequences
`esccharstring) : string` produces the character value of an Icon string escape sequence
`escape()` converts a string with escape sequences (as in Icon string representation) 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`. (REG)

evallist

`evallist(expr, n, ucode, ...) : list` takes an expression, produces a program encapsulating it, and puts the results written by the program in a list. `expr` is an expression (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`. (REG)

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, ... produce values convertible to strings. In the example above, the second argument is a one−character string `" "`, so that `!" "` generates a single blank. (REG)

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`. (REG)

factors

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; default, 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^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 <= 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`. (REG, GMT)

fastfncs

These procedures implement integer−values using the fastest method known to the author. "Fastest" does not mean "fast".

`acker(i, j)`	Ackermann’s function
`fib(i)`	Fibonacci sequence
`g(k, i)`	Generalized Hofstader nested recurrence
`q(i)`	"Chaotic" sequence
`robbins(i)`	Robbins numbers

See also: iterfncs.icn, memrfncs.icn, recrfncs.icn. Links: factors, memrfncs. (REG)

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)`. (REG)

filenseq

`nextseqfilename(dir, pre, ext) : string?` is useful when you need to create the next file in a series of files (such as successive log files). Usage: `fn := nextseqfilename(".", ".", "log")` returns the (non−existent) filename next in the sequence `.\.*.log` (where the * represents 1, 2, 3, ...) or fails Requires: MS−DOS or a more congenial operating system. (DAG)

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 `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. (RLG)

`^` − matches if the following pattern is at the beginning of a line (i.e. `^#` matches lines beginning with `"#"`)

`$` − matches if the preceding pattern is at the end of a line

`.` − matches any single character

+ − matches from 1 to any number of 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 recognized by this program (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 (i.e. abc|cde matches either abc or cde)

[] − matches any member of the enclosed character set, or, if ^ is the first character, any nonmember of the enclosed character set (i.e. [^ab] matches any character *except* a and b).

() − 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")

---

gauss

gauss_random(x, f) produces a Gaussian distribution about the value x. The value of 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 (SBW)

---

gdl, gdl2

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.

gdlrec(dir, findflag) : list does same thing as gdl except it recursively descends through subdirectories. If findflag is nonnull, the UNIX "find" program is used; otherwise the "ls" program is used. Requires: UNIX or MS−DOS. (RLG)

---

gener

These procedures generate sequences of results. (REG)

days() : string* produces the days of the week, starting with "Sunday".
hex() : string* is the sequence of hexadecimal codes for numbers from 0 to 255
label(s,i) : string* produces labels with prefix s starting at i
multii(i, j) : integer* produces 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`, `fastfncs`, `partit`, `numbers`. (REG)

getmail

`getmail(x) : message_record*` reads an Internet mail folder and generates a sequence of records, one per mail message. It fails 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. (CS)

getpaths

`getpaths(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.

Requires: UNIX or MS−DOS. (RLG)

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

(REG)

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. (RJA)

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. (GMT)

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. (RLG)

---

iftrace

These procedures provide tracing for Icon functions by using procedure wrappers to call the functions. iftrace(fncs[]) sets tracing for a list of function names. Links: `ifncs`. (SBW, REG)

---

image

`Image(x, style:1) : string` generalizes the function `image(x)`, providing detailed information about structures. The value of `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. (MG, REG, DY)

---

inbits

`inbits(file,len:integer) : integer` re−imports data converted into writable form by `outbits()`. See also: `outbits.icn`. (RLG)

---

indices

`indices(spec:list, last) : list` produces a list of the integers given by the specification `spec`, which is a comma separated list of positive integers or integer spans, as in `"1,3−10,..."` If `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. (REG)

inserts

`inserts(table,key,value) : 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)

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. (RJA)

io

These procedures provide facilities for handling input, output, and files. Some require `loadfunc()`. Links: `random`, `strings`. (many)

`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.

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.

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.

`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) produces the name of a temporary file.

---

iolib

This library provides control functions for text terminals, primarily ANSI and VT−100 devices under UNIX and MS−DOS. MS−DOS users must add "device=ansi.sys" (or a similar driver) to the config.sys file. The TERM and TERMCAP environment variables must also be set in order to use this library. The TERM variable tells iolib what driver you are using, e.g. TERM=ansi−color for a typical MS−DOS user. The TERMCAP variable gives the location of the termcap database file, as provided on your UNIX system or given in the included file termcap.dos for MS−DOS.

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 not only string−valued sequences, but numeric ones

as well. The value of "li" tells you how many lines the terminal has ("co" tells you 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 as a term. 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 given capability is signaled by procedure 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 or MS−DOS, co−expressions. See also: iscreen.icn. (RLG, NA)

---

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 or MS−DOS. Links: `iolib`. (RLG)

---

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. If `keyproc` is omitted, sorting occurs in the standard Icon order; otherwise, sort keys are obtained from within elements, instead of using the element itself as the key. `keyproc` can be a procedure, in which case the first argument to the key procedure is the item 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 specifies a subscript to be

applied to each item to produce a key. keyproc will be called once for each element of structure x. (RJA)

---

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 that 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. (RLG)

---

ivalue

ivalue(string) : 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. (REG)

---

jumpque

jumpque(queue:list, y) : list moves y to the head of the queue if it is in the queue. Otherwise it adds y to the head of the queue. A copy of the queue is returned; the argument is not modified. (REG)

---

kmap

kmap(string) : string maps uppercase letters and the control modifier key in combination with letters into the corresponding lowercase letters. It is intended for use with graphic applications in which the modifier keys for shift and control are encoded in keyboard events. (REG)

---

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: (1) `s2` is preceded by a character in `c1`, or is found at the beginning of the string; and (2) `s2` is succeeded by a character in `c2`, or the end of the string.

---

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. (REG)

---

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.icn, tab2rec.icn, rec2tab.icn. (REG)

---

lists

Many of these procedures implement list functions similar to string functions, including an implementation of list scanning, similar to Icon's string scanning functions.

`lcomb(L:list,i:integer) : list*` produces all sublist combinations of `L` that have `i` elements.

`ldelete(L:list, spec:string) : list` deletes values of `L` at indices given in `spec`; see `indices`.

`lequiv(L1:list, L2:list) : list?` tests if `L1` and `L2` are structurally equivalent. lequiv() does not detect cycles.

`lextend(L:list, i:integer, x) : list` extends `L` to at least size `i`, using initial value `x`.

`limage(L:list) : string` list image function that shows elements' images, one level deep.

`linterl(L1:list, L2:list) : list` interleaves elements of `L1` and `L2`. If `L1` and `L2` are not the same size, the shorter list is extended with null values to the size of the larger list.

`llpad(L, i, x) : list` produces a new list that extends `L` on its front (left) side.
`lltrim(L:list, S:set) : list` produces a copy of `L` with elements of `S` trimmed on the left.

`lmap(L1,L2,L3) : list` maps elements of `L1` according to `L2` and `L3`, similar to the string−mapping function `map(s1, s2,s3)`. Elements in `L1` that are the same as elements in `L2` are mapped into the corresponding elements of `L3`. For example, given the lists

```
L1 := [1,2,3,4];L2 := [4,3,2,1]; L3 := ["a","b","c","d"]
```

then `lmap(L1, L2,L3)` 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 `map()`, `lmap()` can be used for transposition as well as substitution. **Warning:** If `lmap()` is called with the same lists `L2` and `L3` as in the immediately preceding call, the same mapping is performed, even if the values in `L2` and `L3` have been changed. This improves performance, but it may cause unexpected effects. This "caching" of the mapping table based on `L2` and `L3` can be easily removed to avoid this potential problem.

`lpalin(L:list) : list` produces a list palindrome of `L` concatenated with its reverse.
`lpermute(L:list) : list*` produces all the permutations of list `L`.

`lreflect(L:list, i:0) : list` returns `L` concatenated with its reversal to produce a palindrome. The values of `i` determine "end conditions" for the reversal: 0 = omit first and last elements. 1 = omit first element, 2 = omit last element, 3 = don’t omit element.

`lremvals(L, x1, x2, ...) : list` produces a copy of L, without those elements that are === x1, x2?
lrepl(L:list, i:integer) : list replicates L i times.
lreverse(L:list) : list produces a list that is the reverse of L.
lrotate(L:list, i:integer) : list produces a list with the elements of L, rotated i positions.
lrpad(L:list, i, x) : list is like lextend(), but produces a new list instead of changing L.
lrtrim(L:list, S:set) : list produces a copy of L with elements of S trimmed on the left.
lswap(L:list) : list produces a copy of L with odd elements swapped with even elements.
lunique(L:list) : list 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)	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","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"]]) − 1)
         every writes(!l_tab(0))
         write()
         })
      })
end
```

The above program writes "hello" and "there" on successive lines to the standard output.

The functions compare lists, not strings, so `l_find("h", l)`, for instance, will yield an error message: use `l_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 `l_find()` call needs to look like `l_find([l2],l1)`.

Links: `indices`. See also: `structs.icn`. (REG, RLG)

---

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: MT−Icon, system(), pipes, /tmp. Links: io. (REG)

---

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. (JN,SBW,KW,RJA,RLG)

---

lrgapprx

lrgapprx(i) : string produces an approximate of an integer value in the form i.jx10^k. It is primarily useful for large integers. (REG)

---

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 $M \times X = B$. 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"; William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling; Cambridge University Press, 1986. (REG)

---

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 "010000010111100001100101". Links: strings. (REG)

---

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`. (RLG)

---

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`. (REG)

---

matrix

This file contains procedures for matrix manipulation. Matrices are represented as lists of lists.

`matrix_width(M:list) : integer` produces the number of columns in a matrix.
`matrix_height(M:list) : 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`.

Links: `lu`. (SBW, REG)

---

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. (GMT)

---

morse

`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). (REG, RM)

---

mset

`same_value(d1,d2) : null?` compares `d1` and `d2` for structural equivalence.
`insert2(S:set, el) : set` inserts `el` into `S`

`member2(S:set, el) : null?` tests whether `el` (or its structural equivalent) is in `S`.
`delete2(S:set, el) : set` deletes `el` (or its structural equivalent) from `S`.

The mset module implements set operations in which no two identical (structurally equivalent) values can be present in a set. (JPR)

---

namepfx

`namepfx(s:string) : string` produces the "name prefix" from a name in standard form −− omitting any title, but picking up the first name and any initials. `namepfx()` only knows how to omit common titles found in module `titleset`. Obviously, it can't always produce the "correct" result. Links: `lastname`, `titleset`. (REG)

---

ngrams

`ngrams(file,n,c:&letters,t) : string*` generates a tabulation of the n−grams in the specified file. If `c` is non−null, it is used as the set of characters from which n−grams are taken (other characters break n−grams). If `t` is non−null, the tabulation is given in order of frequency; otherwise in alphabetical order of n−grams. Note: This procedure is unsuitable if there are very many different n−grams. (REG)

---

numbers

These procedures format numbers in various ways:

`amean(L:list) : real` returns arithmetic mean of numbers in L.
`ceil(r:real) : integer` returns nearest integer to `r` away from 0.
`commas(s:string) : string` inserts commas in s to separate digits into groups of three.
`decipos(r:real, i:3, j:5) : string?` positions decimal point at `i` in `r` in field of width `j`.
`digred(integer) : integer` reduces a number by adding digits until one digit is reached.
`div(i:number, j:number) : real` produces the result of real division of `i` by `j`.

`fix(i, j:1, w:8, d:3) : string?` formats `i / j` as a real (floating−point) number in a field of width `w` with `d` digits to the right of the decimal point, if possible. If `w` is less than 3 it is set to 3. If `d` is less than 1, it is set to 1. The function fails if `j` is 0 or if the number cannot be formatted.

`floor(r) : integer` nearest integer to `r` toward 0
`gcd(i:integer, j:integer) : integer?` returns greatest common divisor of i and j. It fails if both are zero.
`gcdl(L:list) : integer` returns the greatest common division of a list of integers.
`gmean(args?) : real?` returns the geometric mean of numbers.
`hmean(args?) : real?` returns the harmonic mean of numbers.
`large(i) : integer?` succeeds if `i` is a large integer but fails otherwise.

`lcm(i, j) : integer?` returns the least common multiple of i and j.
`lcml(L) : integer?` returns the least common multiple of the integers in the list L.
`max(args?) : number` produces the maximum of its argument numbers.
`min(args?) : number` produces the minimum of its argument numbers.
`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.

Links: `strings`. (REG, RJA, RLG, TK)

---

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. (DAG)

---

options

`options(args:list, opt:string, err:stop) : table` separates, interprets, and removes UNIX−style command options from the argument list of an Icon `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 appear in combination, 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: `"−Ffile.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. (RJA, GMT)

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. (RLG)

---

packunpk

pack(num:integer, width:integer) : string produces a binary−coded decimal representation of num in which each character contains two decimal digits stored in four bits each.

unpack(val:string, width:integer) : string converts a binary−coded decimal back to a string representation of the original source integer. The string is formatted in width characters.

`unpack2(val:string) : integer` converts a binary−coded decimal back into its original source integer.

Links: `convert.` (CT, RLG)

---

partit

`partit(i:integer, min:integer, max:integer) : list*` generates the partitions of `i`; that is the ways that `i` can be represented as a sum of positive integers with minimum and maximum values.

partcount(i:integer, min:integer, max:integer) : integer returns the number of partitions.

`fibpart(i:integer) : list` returns a list of Fibonacci numbers that is a partition of `i`.

Links: `fastfncs`, `numbers`. (REG)

---

patterns

This module provides string scanning procedure equivalents for most SNOBOL4 patterns and some extensions. Procedures and their pattern equivalents are:

Procedure	Equivalent SNOBOL pattern
Any(s)	ANY(S)
Arb()	ARB
Arbno(p)	ARBNO(P)
Arbx(i)	ARB(I)
Bal()	BAL
Break(s)	BREAK(S)
Breakx(s)	BREAKX(S)
Cat(p1,p2)	P1 P2
Discard(p)	/P
Exog(s)	\S
Find(s)	FIND(S)
Len(i)	LEN(I)
Limit(p,i)	P \ I
Locate(p)	LOCATE(P)
Marb()	longest−first ARB
Notany(s)	NOTANY(S)
Pos(i)	POS(I)
Replace(p,s)	P = S
Rpos(i)	RPOS(I)
Rtab(i)	RTAB(I)
Span(s)	SPAN(S)
String(s)	S
Succeed()	SUCCEED
Tab(i)	TAB(I)
Xform(f,p)	F(P)

The following procedures relate to the application and control of pattern matching:

Apply(s,p)	S ? P
Mode()	anchored or unanchored matching (see Anchor() and Float())
Anchor()	&ANCHOR = 1 if Mode := Anchor
Float()	&ANCHOR = 0 if Mode := Float

In addition to the procedures above, the following expressions can be used:

p1() | p2()	P1 | P2
v <− p()	P . V (approximate)
v := p()	P . V (approximate)
fail	FAIL
=s	S (in place of String(s))
p1() || p2()	P1 P2 (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

```
SPAN("0123456789") . N "H" LEN(*N) . LIT
```

can be transliterated into

```
(n <− Span('0123456789')) || ="H" || (lit <− Len(n))
```

Concatenation of components is necessary to preserve the pattern−matching properties of SNOBOL4. (REG)

---

patword

`patword(s:string) : string` returns a letter pattern in which each different character in `s` is assigned a letter. For example, `patword("structural")` returns `"abcdebdcfg"`. (KW)

---

phoname

`phoname(telno:string) : 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. (TRH)

---

plural

`plural(word:string) : string` produces the plural form of a singular English noun. The procedure here is rudimentary and is not correct in all cases. (REG)

---

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)

`poly(c1, e1, c2, e2, ...) : poly` creates a polynomial from the parameters given as coefficient−exponent pairs: `c1x^e1 + c2x^e2 + ...`
`is_zero(n) : null?` determines if n = 0

`is_zero_poly(p) : null?` determines if a given polynomial is 0x^0
`poly_add(p1, p2) : poly` returns the sum of two polynomials
`poly_sub(p1, p2) : poly` returns the difference of p1 − p2
`poly_mul(p1, p2) : poly` returns the product of two polynomials
`poly_eval(p, x) : poly` finds the value of polynomial p when evaluated at the given x.
`term2string (c, e) : string` converts one coefficient−exponent pair into a string.
`poly_string(p) : string` returns the string representation of an entire polynomial.

---

printcol

printcol(items, fields, title:"", pagelength:30000, linelength:80, 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). (RJA)

For example, suppose you have a list of records to print. The record is defined as:
`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 item1 ...
Field 2 item2 item2 ...
Field 3 item3 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 that can be produced (including the column header).

`other` produces the field value string for the item passed as the argument.

The second argument is arbitrary data from the procedures with each invocation. The data returned by the first function on the list is used as a column heading string (the item name).

`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 arguments and writes them to `&output`.
`fprintf(f:file, fmt:string, args[])` formats arguments and writes them 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. (WHM, CW, PLT)

---

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. (REG)

---

procname

`procname(p:procedure, x) : string?` produces the name of a procedure from 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. (REG)

---

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. Usage example:

```
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")
```

(GMT)

---

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 to a "random" value, based on the time of day and the date.
randrange(min, max) : integer produces random number in the range 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 allocating a new structure to hold the shuffled result.

Links: factors. (REG, GMT)

---

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. (REG)

---

readtbl

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. (RLG)

---

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.) (REG)

---

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. (REG)

---

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. (REG)

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. (REG)

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 (see notes, below). String scanning function default conventions are followed.

`ReMatch(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.

`ReFind(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()`. 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()`, below.

`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 sets of characters (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 ' \t\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 procedure 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 procedure 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. In the description, 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 `"^"` (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 at least N occurrences but at most 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 the same string of characters as 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 only at a word−boundary (word defined as 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.

(RJA)

---

repetit

`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]
```

`repetit(L)` returns 3. If there is no repetition, `repetit()` returns the length of the list. (REG)

---

rewrap

`rewrap(s:string,i:70) : string?` reformats text fed to it into strings `< i` in length. `rewrap()` utilizes a static buffer, so it can be called repeatedly with different string arguments, and still produce homogenous output. This buffer is flushed by calling `rewrap()` with a null first argument. Here's a simple example of how `rewrap()` could be used. The following program reads the standard input, producing fully rewrapped output.

```
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

```
procedure main()
   while line := read(&input) do {
      if line == "" then {
         write("" ~==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:

```
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`. (RLG)

---

scan

This module contains procedures related to string scanning: Where indicated by parameter names, they follow the string scanning conventions for parameter defaults.

`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 as defined in `c5`.

`balqc(c1,c2,c3,c4:'\'"',c5,s1:"/*",s2:"*/",s3,i1,i2):integer*` is like `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.

`limatch(L:list, c:cset) : integer?` matches items in L delimited by characters in c. Returns the last cursor position scanned to, or fails

`slashbal(c1,c2,c3,s,i,j) : integer*` behaves like `bal()`, except that it ignores, for purposes of balancing, any `c2` or `c3` char which is preceded by a backslash.

`slashupto(c, s, i, j) : integer?` works just like `upto()`, except that it ignores backslash escaped characters.

`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'`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.

(RLG, DAG, REG, RLS, CW)

---

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
```

(RJA)

---

segment

These procedures segment a string s into consecutive substrings consisting of characters that respectively do/do not occur in c.

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","."]

(WHM)

---

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" that is detailed in the module source code. (RLG, PAB)

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 not omitted and `e` has an infinite result sequence, `Seqimage{}` does not terminate. (REG)

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. (REG)

sets

`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`.

(AB, REG)

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:

`mode` indicates the type of sorting, one of: `"ref" | "val"` (by key or by 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. (REG)

shquote

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 (MPW 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.

(RJA)

signed

`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. (RJA)

sort

`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 specified 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. (RJA,RLG,REG)

soundex, soundex1

`soundex(name:string) : string` produces a code for a name that tends to bring together variant spellings. See Donald E. Knuth, The Art of Computer Programming, Vol.3; Searching and Sorting, pp. 391−392. (CW)

Module `soundex1` employs an approach proposed by Margaret K. Odell and Robert C. Russell; see the source code for details. (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. (REG)

str2toks

String 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). Naturally, 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. (RLG)

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 the combinations of characters from s taken i at a time.
`compress(s, c:&cset) : string` compresses consecutive occurrences of characters in c that occur 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` interleaves 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 first appear in `s`.
`palins(s, n) : string*` generates all the `n`−character palindromes from the 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 replaced all a's by `"bc"` and all d's by `"cd"`. Replacements are performed one after another, not in parallel.

`replc(s, L) : string` replicates each character of `s` by the amount given by the values in `L`.
`rotate(s, i:1) : string` rotates `s` by `i` characters to the 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` according to label `s2` and transposition `s3`.

(REG)

---

stripcom

`stripcom(s) : string?` strips the commented−out portion of a line of Icon 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. (RLG)

---

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", "\e1m") 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, off)`. When `"</>"` is encountered, `stripunb()` will output the `.off` value for the preceding `.on` string encountered.

Links: `scan`. (RLG)

---

tab2list, tab2rec

`tab2list(string) : list` takes tab−separated strings and inserts them into a list. Vertical tabs in strings are converted to carriage returns.

`tab2rec(s, x) : null` 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. (REG)

---

tables

`keylist(T) : list` produces a list of keys in table `T`.
`kvallist(T) : list` produces values in `T` ordered by sorted order of keys.
`tbleq(T1, T2) : table?` tests the equivalence of tables `T1 and T2`.
`tblunion(T1, T2) : table` approximates `T1 ++ T2`.
`tblinter(T1, T2) : table` approximates `T1 ** T2`.
`tbldiff(T1, T2) : table` approximates `T1 −− T2`.
`tblinvrt(T) : table` produces a table with the keys and values of `T` swapped.
`tbldflt(T) : any` produces the default value for `T`.

`twt(T) : table` produces a two−way table based on `T`.
`vallist(T) : list` produces list of values in table `T`.

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. (REG, AB)

---

tclass

`tclass(x) : string` returns "atomic" or "composite" depending on the type of `x`. (REG)

---

title, titleset

`title(name:string) : string` produces the title of a name, such as "Mr." from "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. (REG)

---

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 that use a list representation of trees and directed acyclic graphs (DAGs). These procedures do not protect themselves from cycles. (REG)

---

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([exrp1, 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
   }
```

(WHM)

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. (REG)

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. (RJA)

usage

These procedures provide various common services: (REG)

`Usage(s)` stops execution with a message concerning the expected usage of a program.
`Error(args?)` writes arguments to `&errout` on a single line, preceded by `"*** "`.
`ErrorCheck(l,f) : null` reports to `&output` an error that has been converted to failure.
`Feature(s) : null?` succeeds if feature `s` is available in the running implementation of Icon.
`Requires(s)` terminates execution is 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)

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. (REG)

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.

Links: `records`. Requires: Version 9 graphics for color conversion. (REG)

---

wdiag

`wdiag(s1, s2,...) : null` writes the values of the global variables named `s1, s2, ...` with `s1, s2, ...` as identifying labels. It writes a diagnostic message to standard error output if an argument is not the name of a global variable. Note that this procedure only works for global variables. (REG)

---

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
`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. (EE)

---

wildcard

This is a kit of procedures to deal with UNIX−like filename wild−card patterns containing *, ?, and [...]. The meanings are as of the pattern characters are the same as in the UNIX shells csh and sh. They are described briefly in the `wild_pat()` procedure. These procedures are interesting partly because of the "recursive suspension" technique used to simulate conjunction of an arbitrary number of computed expressions. 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.

Default values of `s`, `i1`, and `i2` are the same as for Icon's built−in string scanning procedures such as `match()`.

`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. (RJA)

---

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`. (RJA)

---

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.
```

`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.

(RJA)

---

xcodes

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.

These procedures provide a way of storing Icon values in files and retrieving them. xencode(x,f) stores x in file f such that it can be converted back to x by xdecode(f). These procedures handle several 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, the above sequence will result in the relationship x === y. For structured types −− list, set, table, and record types −− 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 cannot generally be restored in any way that makes much sense; 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 it 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:

```
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. (RJA, REG)

---

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

```
   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 ximage() infers the predominant list element value and avoids excessive output):

```
"A table"
T18 := table(&null)
T18["one"] := 1
T18["two"] := 2
"A list"
L25 := list(8,3)
L25[2] := 1
L25[4] := L24 := list(3)
L24[1] := 2
L24[2] := 4
L24[3] := 6
L25[6] := 4
L25[8] := 5
```

(RJA)

xrotate

`xrotate(X, i)` rotates the values in `X` right by one position. It works for lists and records. This procedure is mainly interesting as a recursive version of `x1 :=: x2 :=: x3 :=: ... xn` since a better method for lists is `push(L, pull(L))`. (REG)

GUI Classes

This section presents the various methods and class fields making up the classes implemented in the Unicon GUI library, `gui.iol`. 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.

Generally speaking, if a required field is omitted then an error message may be produced, such as the following:

```
gui.iol : error processing object TextList : x position
unspecified
```

This means that the `x` position of a `TextList` object was not specified, probably because the `set_pos()` method had not been invoked.

GUI classes depend on graphics facilities defined in "Graphics Programming in Icon", by Griswold, Jeffery, and Townsend. You will generally need to consult that book in order to understand how to use these classes.

_Event

An instance of this class is generated by components and passed to the dialog_event method. It simply holds three elements, which can be accessed as follows:

`get_event()` returns the Icon event associated with the event.

`get_component()` returns the component associated with the event. This may be a subclass of either `Component` or a `MenuComponent`. If this is `&null`, then an Icon event has occurred which has not produced an `_Event` from any component.

`get_code()` returns an additional field to distinguish between different types of event generated by the same component. For example, a `TextField` produces a code of zero when return is pressed, and a code of one when the contents are changed.

_Dialog : Container

This is the parent class of a dialog window.

`set_initial_focus(c)` sets the initial keyboard focus to the given component. The given component will then have the keyboard input focus when the window is displayed.

`set_is_modal()` causes the `pending()` method to be repeatedly invoked while the `Dialog` is waiting for an event to occur. The only exception is that `pending()` will not be invoked while a drop−down list or menu is open.

`pending()` is repeatedly invoked between window events, for modal dialogs only. This is used to perform extra processing while the program is waiting for the next event.

`resize_win(w, h)` resize the window to the given dimensions.

`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 the given component.
`clear_focus()` clears the keyboard focus.

`show()` opens and displays the dialog window, and accepts events into it. As events occur they are passed to `dialog_event()`, until a call to `dispose()` is made. Then the window is closed and the method returns.

`dialog_event(e:_Event)` must be over−ridden in the subclass. It is the method that is invoked when an event occurs.

`dispose()` is normally invoked from within `dialog_event()`. It sets a flag to indicate that the dialog should be closed.

`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()` opens and displays the dialog window. The method just calls other methods in sequence, as follows:

```
method show()
   Open()
   final_setup(self, self)
   resize()
   firstly()
   is_open := 1
   display()
   set_focus(\initial_focus)
   process_events()
   finally()
   Close()
   return
end
```

Note that some of these methods are those inherited from `_Dialog`'s parent class, `Container`.

`Open()` opens the dialog window.
`Close()` closes the dialog window.

`process_events()` implements the event handling loop of _Dialog.

`win : window` is the dialog's window.
`is_open : flag` indicates whether the window is open.

`done : flag` indicates whether dialog should close on the next iteration of the toolkit's event loop; set by `dispose()`.

`initial_focus : Component` specifies the component having initial keyboard focus; set by `set_initial_focus()`.

`focus : Component` specifies the component with the current focus.

`unique : flag` controls unique processing mode, in which one component alone will receive all events.

`re_process_flag : flag` indicates 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.
`is_modal_flag : flag` indicates whether to call `pending()` method between events; set by `set_is_modal()`.

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()` is called by an object whose reference variable is set to this object to get the `x` value from which to compute absolute positions. May be over−ridden, but by default just returns `self.x`.

`get_y_reference()` is the same as above for the `y` position.
`get_w_reference()` is the same as above for the width
`get_h_reference()` is the same as above for the height.

`get_cwin_reference()` is the same as above for the cloned window (the reference object inherits the attributes by cloning this window).

`get_cbwin_reference()` is the same as above for the cloned buffer window.
`error(s)` prints an error message `s` together with the class name, and stops the program.
`redisplay()` calls `display()` if and only if the dialog window is open.
`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.

`generate_components() : Component*` generates the components for the object. By default this just returns the component itself.

`toggle_is_shaded()` swaps the shaded status of the component.
`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_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.

`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.

`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 .

`set_attribs(x[])` sets the Icon attributes of the component to the given parameter list. For example: `w.set_attribs("font=helvetica", "bg=pale blue")`

set_attribs_list(l) is like set_attribs(), but takes a list as a parameter. For example: w.set_attribs_list(["font=helvetica", "bg=pale blue"])

`get_parent_win() : window` returns the Icon window of the dialog 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 can be given in the form of a percentage plus or minus an offset. Examples:

```
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 so that its center is in the
# center of the window.
c.set_pos("50%", "50%")
c.set_align("c", "c")

# Place c so that its top left corner is
# at position (10, 10).
c.set_pos(10, 10)
c.set_align("l", "t")
```

`set_size(w, h)` sets the size of the component. The parameters are in the same format as for `set_pos()` above. Some components will set sensible default sizes, but for 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 draw into (normally cwin).

`in_region()` succeeds if the component is not shaded and the values of `&x` and `&y` lie 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.

`final_setup(x, y)` sets the component's `parent_Dialog` to `x`, and its reference variable to `y`. It then 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 just before the window is closed. This method may be extended in subclasses. For example:

```
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` is the x position as specified by `set_pos()`, e.g. `"50%"`
`y_spec` is the y position as specified by `set_pos()`.
`w_spec` is the width specifier as specified in `set_size()`, e.g. `"100%"`
`h_spec` is the height specifier as specified in `set_size()`,
`x_align` is the x alignment as specified in `set_align()`, e.g. `"l"`.
`y_align` is the y alignment as specified in `set_align()`, e.g. `"b"`.

The following four attributes are absolute dimensions in pixels, compiled from `x_spec`, `y_spec`, `w_spec`, and `h_spec`, and the dimensions of the enclosing object or window.

`x` is the x position computed from `x_spec`.
`y` is the y position computed from `y_spec`.
`w` is the width computed from `w_spec`.
`h` is the height computed from `h_spec`.

`parent_Dialog` is the `_Dialog` class instance of which this `Component` is a part.

`attribs` is a list of strings specifying Icon graphics attributes, e.g. `["bg=blue", "resize=on"]`.

`has_focus : flag` indicates whether the `Component` currently has the keyboard focus.

`is_shaded_flag : flag` indicates whether the `Component` currently is shaded.

`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.

`cbwin` is a cloned window created by combining a buffer window with the `Component`'s attributes. This is used solely for double−buffering purposes.

`accepts_tab_focus_flag : flag` indicates whether the `Component` accepts keyboard focus by way of the tab key being pressed.

`draw_border_flag : flag` indicates whether the `Component` should have a border drawn around it. Many components (such as `TextButtons`) ignore this flag.

`reference` links to the object that is used to calculate absolute sizes from percentage sizes. For objects placed directly into the `_Dialog`, rather than into some other object, this will point to the `_Dialog` instance itself, which over−rides the several methods `get_x_reference()` etc., appropriately.

Container : Component

This class acts as a container for other components. The component itself is not displayable. Many of `Component`'s methods are over−ridden by this class. A `_Dialog` instance is a sub−class of this class.

`add(c:Component)` adds the component `c` to the `Container`.

`components` are the components inside the `Container`.

VisibleContainer : Component

This is similar to a `Container`, except that the object itself is a capable of display.

add(c:Component) adds the component c to the VisibleContainer.

`components` are the components inside the `VisibleContainer`.

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. Then, when return is pressed an event of code 2 is generated. The method `_Dialog.set_initial_focus()` can be used to give the button the keyboard focus when the dialog is first displayed.

`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.

TextButton : Button

A button with a text label. The size of the button can either be set using `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~~~~_
~~0000000~~_
~000000000~_
~~0000000~~_
~~~~000~~~~_
~~~~~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.

NB − A `Button` must be added to the `ButtonGroup` and the `Dialog` too. Examples:

```
bg := ButtonGroup()
b := TextButton()
b.set_label("Okay")
add(b)
bg.add(b)
```

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()`. 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 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()` causes the image to be scaled up to fit in the space specified by `set_size()`. The image will not be 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 always be scaled down.

`set_internal_alignment(x:"c", y:"c")` sets the horizontal and vertical alignment of the image within the area of the component; `x` should be `"l"`, `"c"` or `"r"`, `y` should be `"t"`, `"c"` or `"b"`.

Border : VisibleContainer

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 add(c) method from the parent class VisibleContainer is used to set the title.

`set_internal_alignment(x)` sets the alignment of the title object. The input string should be `"l"`, `"c"` or `"r"`. Examples:

```
b := Border()
#
# Add a Label as the title
#
l := Label()
l.set_label("Title String")
b.add(l)
add(b)
```

ScrollBar : Component

This class provides horizontal and vertical scroll bars. There are two ways to use a scroll bar. The first way 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 will then range from zero to (`total_size - page_size`) inclusive. An initial value must be set with the `set_value()` method. Examples:

```
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 that ranges 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`, as above, 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. Examples:

```
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 range of the scroll bar from x to y inclusive; x may be integer or real.

`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 to increase the value by when one of the buttons 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 of 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:

```
t := TextField()
```

```
t.set_pos(50, 250)
# Vertical size will default
t.set_size(100)
t.set_contents("Initial string")
add(t)
```

CheckBox : Component

An instance of this class is 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 will default to appropriate values if not specified. The size will also default 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:

```
c := CheckBox()
c.set_pos(200, 100)
c.set_label("Checkbox")
add(c)
```

CheckBoxGroup

This class contains several `CheckBox` objects that act together as "radio buttons". Adding a `CheckBox` to a `CheckBoxGroup` sets the image styles of the `CheckBox` to 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:

```
#
# 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_contents(x:list)` sets the contents to `x`, which should be a list of strings.

`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, 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. Examples:

```
tl := TextList()
tl.set_contents(data)
 ...
# Amend data and go to end of data
put(data, "New line")
tl.set_contents(data, *data, 0)
```

`set_select_one()` specifies that only one line of the list may be highlighted. Of no effect if in editable mode.

`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 line numbers that are selected using the given list of line numbers. Examples:

```
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()` returns an integer corresponding to the item in the list presently selected.

`set_initial_selection(x)` sets the index of the initial item in the list to be displayed; the default is the first if this method is not invoked.

List : Component : DropDown

This component is for selecting one string from a list of several. 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 from the list.

`set_constant_label(x)` supplies a string that will always appear in the text part of the component, rather than the currently selected item.

A `width` must be specified for this component.

The methods that are needed to handle the list of selections are inherited from DropDown. Examples:

```
L := List()
L.set_selection_list(["Red", "Green", "Yellow", "Blue",
     "Orange"])
L.set_size(120)
L.set_pos(100, 100)
L.set_initial_selection(2) # Green will be the first selection
add(L)
```

EditList : Component : DropDown

An EditList functions in the same way as a `List` above, 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, to `x`.

MenuEvent : _Event

This is just a subclass with an extra code variable.

`menu_code` is returned by a `MenuComponent` and indicates to the menu system what to do with the event that has been produced. It takes one of the following values:

Code	Action
FAIL_1	The Menu system closes and the last event is not passed on to the dialog.
FAIL_2	The Menu system closes and the last event is passed on to the dialog.
CONTINUE	The Menu system does not close and continues to process events.
SUCCEED	The menu system closes and produces the given event, which will be passed to the dialog's `dialog_event()` method. The last event will not be passed on to other components.

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 itself.

`hide_non_menu()` is empty; it is implemented by the `Menu` class and is called for any `SubMenu` object, but would not normally need to be over−ridden by a custom class.

`set_which_open(x)` is another empty method. The same comment applies to this method.

`resize()` is another empty method, but it may be convenient to over−ride it in order to initialize the width and height of the object here.

`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 added to 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 simply a horizontal bar in a `Menu`, for decorative purposes. It has no methods that the user need invoke.

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 must be specified. Examples:

```
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. If not invoked, a sensible default will be used.

`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, 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.

`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:list)` 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 can be 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 TabItem that is to be displayed first. The default is the first that was added.

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 attributes.

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 which OverlayItem is currently displayed. The default is the first that was added.

add(c:OverlayItem) adds the given OverlayItem to the OverlaySet.

Application Programs, Examples, and Tools

The Icon Program Library progs directory contains 200+ programs that are useful for demonstration, entertainment, and/or practical utility.

---

adlcheck, adlcount, adlfilter, adlfirst, adllist, adlsort

adlcheck checks address lists for correctness. There are five options:

`-s` check state (U.S. labels only)
`-z` check ZIP code (U.S. labels only)
`-c` check country name (a very heuristic check)
`-a` check all of the above
`-d` report addresses that exceed "standard dimensions" for labels: 40 character line length, 8 lines per entry

`adlcount` counts the number of entries in an address list file. If an argument is given, it counts only those that have designators with characters in the argument. Otherwise, it counts all entries.

`adlfilter` filters address lists, allowing through only those entries with specified selectors. The option `-s arg` selects entries with characters in args (default is all). Option `-x` inverts the logic, selecting characters not in args

`adlfirst` writes 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. Otherwise, it counts all entries.

`adllist` lists entries in address lists. The options are:

`-c` by country
`-n` by name (default)
`-C` by city (U.S. only)
`-s` by state (U.S. only)
`-z` by ZIP code (U.S. only).

If more than one option is specified, the order of dominance is `-n -z -s -c -C`.

`adlsort` sorts entries in address lists. The options are:

`-c` by country
`-n` by name (default)
`-z` by ZIP code

If more than one option is specified, the order of dominance is `-n -z -c`.

See also: address.doc, labels.icn. Links: adlutils, options, namepfx. (REG)

---

animal

`animal` is the familiar "animal game" written in Icon. The program asks its human opponent a series of questions in an attempt to guess what animal he or she is thinking of. It is an "expert system" that starts out with limited knowledge, knowing only one question, but gets smarter as it plays and learns from its opponents. At the conclusion of a session, the program asks permission to remember for future sessions that which it learned. The saved file is an editable text file, so typos entered during the heat of battle can be corrected. The game is not limited to guessing only animals. By simply modifying the first two lines of procedure "main" a program can be created that will happily build a knowledge base in other categories. For example, the lines:

```
GameObject := "president"
Tree := Question("Has he ever been known as Bonzo",
```

```
"Reagan","Lincoln")
```

can be substituted, the program works reasonably well, and could even pass as educational. The knowledge files will automatically be kept separate, too. Typing "list" at any yes/no prompt will show an inventory of animals known, and there are some other commands too. (RJA)

---

banner

`banner` is a utility inspired by the UNIX `banner(1)` command. It outputs enlarged letters (5x6 matrix) in portrait mode. With a little fiddling you can change the scale or font. (CT)

---

bj

`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.

Links: `iolib`, `random`. Requires: UNIX. (CT, RLG)

---

blnk2tab

`blnk2tab` converts strings of two or more blanks to tabs. It reads from standard input and writes to standard output. (REG)

---

c2icn

The `c2icn` filter does some of the mundane work involved in porting a C program to Icon. It reformats comments, moving embedded comments to end of line, removes the ";" from ends of lines, reformats line−continued strings, changes `=` to `:=`, and changes `->` to `.` (RJA)

---

calc

`calc` is a simple Polish "desk calculator". It accepts as values Icon integers, reals, csets, and strings (as they would appear in an Icon program) as well as an empty line for the null value. Other lines of input are interpreted as operations. These may be Icon operators, functions, or the commands listed below. In the case of operator symbols, such as `+`, that correspond to both unary and binary operations, the binary one is used. Thus, the unary operation is not available. In case of Icon functions like `write()` that take an arbitrary number of arguments, one argument is used. The commands are:

clear	remove all values from the calculator’s stack
dump	write out the contents of the stack
quit	exit from the calculator

Failure and most errors are detected, but in these cases, arguments are consumed and not restored to the stack. Links: `ivalue`, `usage`. (REG)

---

chkhtml

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.
* Bad (non−ASCII) characters

The program also advises on the use of `<HTML>`, `<HEAD>`, and `<BODY>` tags. (RJA)

---

colm

`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.

Usage: colm [−w line_width] [−s space] [−m min_width] [−t tab_width] [−x] [−d] [file ...]

`linewidth`: the maximum width allowed for output lines (default: 80).
`space`: minimum number of spaces between items (default: 2).
`min_width`: minimum width to be printed for each entry (default: no minimum).
`tab_width`: tab width used to entab output lines. (default: no tabs).

`−x` print items in row−wise order rather than column−wise.
`−d` distribute columns throughout available width.

The command `colm −h` generates a help message. (RJA)

---

comfiles

`comfiles` lists common file names in two directories given as command−line arguments. Requires: UNIX (REG)

concord

`concord` produces a simple concordance from standard input to standard output. Words less than three characters long are ignored. There are two options:

`−l n` set maximum line length to n (default 72), starts new line
`−w n` set maximum width for word to n (default 15), truncates

Links: `options`. (REG)

conman

`conman` responds to queries like "? Volume of the earth in tbsp". The keywords of the language (which are not reserved) are:

load save print ? (same as print) list is are (same as is)

"Load" followed by an optional filename loads definitions of units from a file. If filename is not supplied, it defaults to `conman.sav`.

"Save" makes a file for "load". Filename defaults to `conman.sav`. "Save" appends to an existing file so a user needs to periodically edit his save file to prune it back.

"Print" and "?" are used in phrases like: ? 5 minutes in seconds. Conman replies: 5 minutes in seconds equals 300.

List puts up on the screen all the defined units and the corresponding values. The format is same as load/store format.

"Is" and "are" are used like this: 100 cm are 1 meter The discovery of is or are causes the preceding token (in this case "cm") to be defined. The load/store format is: *unitname* "is" *value*. Examples:

```
8 furlongs is 1 mile
furlong is 1 / 8 mile
```

These last two are equivalent. Note spaces before and after "/".

Continuing examples:

```
1 fortnight is 14 days
furlong/fortnight is furlong / fortnight
inches/hour is inch / hour
```

After this a user might type: ? 1 furlong/fortnight in inches/hour. Conman will reply: 1 furlong/fortnight in inches/hour equals 23.57. Conman’s operators have no precedence so the line above gets the right answer but 1 furlong/fortnight in inches/hour gets the wrong answer. (WED)

countlst

`countlst` counts the number times each line of input occurs and writes a summary. With no option, the output is sorted first by decreasing count and within each count, alphabetically. With the option `−a`, the output is sorted alphabetically. The option `−t` prints a total at the end.

Links: `adlutils`, `options`. (REG)

cross

`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. (WPM)

crypt

`crypt` is an example encryption program. Do *not* use this in the face of competent cryptanalysis.

Usage: crypt [key] < infile > outfile

As written, `crypt` uses UNIX−style console I/O. (PB, PLT)

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 considered to be comments and are ignored. A line consisting of a nonterminal symbol followed by a colon and a nonnegative integer i is a generation specification for i instances of sentences for the language defined by the nonterminal (goal) symbol. An example of input to `csgen` is:

```
# a(n)b(n)c(n)
# Salomaa, p. 11.
# Attributed to M. Soittola.
#
X−>abc
X−>aYbc
Yb−>bY
Yc−>Zbcc
bZ−>Zb
aZ−>aaY
aZ−>aa
X:10
```

The output of `csgen` for this example is

```
aaabbbccc
aaaaaaaaabbbbbbbbbccccccccc
abc
aabbcc
aabbcc
aaabbbccc
aabbcc
abc
aaaabbbbcccc
aaabbbccc
```

A positive integer followed by a colon can be prefixed to a production to replicate that production, making its selection more likely. For example, `3:X−>abc` is equivalent to

```
X−>abc
X−>abc
X−>abc
```

One option is supported: `−g i` number of derivations; overrides the number specified in the grammar. Limitations: Only single uppercase letters may represent nonterminal symbols, and there is no way to represent uppercase letters as terminal symbols. There can be only one generation specification and it must appear as the last line of input. Comments: Generation of context−sensitive strings is a slow process. It may not terminate, either because of a loop in the rewriting rules or because of the progressive accumulation of nonterminal symbols. The program avoids deadlock, in which there are

no possible rewrites for a string in the derivation. This program would be improved if the specification of nonterminal symbols were more general, as in `rsg`.

Links: `options`, `random`. (REG)

---

cstrings

`cstrings` prints all strings (enclosed in double quotes) in C source files. (RJA)

---

cwd

`cwd` writes the current working directory, shorn of its path specification. For appropriately named directories, it can be used as, for example, ftp ‘cwd‘

Requires: UNIX. (REG)

---

daystil

`daystil` calculates the number of days between the current date and the date specified on the command line, and writes this number to &output. This is useful if you want to know how many days it is until a birthday, wedding day, etc. The date on the command line can be specified in a variety of ways. For instance, if you wanted to know how many days it is until August 12, you could specify it as "August 12", "Aug 12", "12 August", or "12 aUGuS", among others. The match is case insensitive, and the arguments will be accepted as long as exactly one of them is an integer, and if there are exactly two arguments. (NL)

---

deal

`deal` shuffles, deals, and displays hands in the game of bridge. An example of the output of deal is

```
--------------------------------
          S: KQ987
          H: 52
          D: T94
          C: T82
S: 3                 S: JT4
H: T7                H: J9863
D: AKQ762            D: J85
C: QJ94              C: K7
          S: A652
          H: AKQ4
          D: 3
          C: A653
--------------------------------
```

Options: The following options are available:

`-h n` Produce n hands. The default is 1.
`-s n` Set the seed for random generation to n. Different seeds give different hands. The default seed is 0.

Links: `options`, `random`. (REG)

declchck

`declchck` examines ucode files and reports declared identifiers that may conflict with function names.

Requires: UNIX. (REG)

delamc

`delamc` delaminates standard input into several output files according to the separator characters specified by the string following the −t option. It writes the fields in each line to the corresponding output files as individual lines. If no data occurs in the specified position for a given input line an empty output line is written. This insures that all output files contain the same number of lines as the input file. If − is used as an output file name, the corresponding field is written to the standard output. If the `−t` option is not used, an ASCII horizontal tab character is assumed as the default field separator. The use of delamc is illustrated by the following examples.

```
delamc labels opcodes operands
```

writes the fields of standard input, each of which is separated by a tab character, to the output files `labels`, `opcodes`, and `operands`.

```
delamc −t: scores names matric ps1 ps2 ps3
```

writes the fields of standard input, each of which are separated by a colon, to the indicated output files.

```
delamc −t,: oldata f1 f2
```

separates the fields using either a comma or a colon.

Links: `usage`, `delamrc`. (TRH)

detex

`detex` reads in documents written in the LaTeX typesetting language, and removes some of the common LaTeX commands to produce plain ASCII. This program is not a full LaTeX parser, and output must typically be further edited by hand to produce an acceptable result. (CLJ)

diffn

`diffn` shows the differences between n files. Usage: `diffn file1 file2 ... filen`

Links: `dif`. (RJA)

diffsort

Usage: diffsort [file]

`diffsort` reorders the output from the Unix "diff" program by moving one−line entries such as "Common subdirectory..." and "Only in ..." to the front of the output file and sorting them. Actual difference records then follow, in the original order, separated by lines of equal signs. (GMT)

diffsum

Usage: diffsum [file]

`diffsum` reads a file containing output from a run of the Unix `diff(1)` utility. Diffsum handles 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. (GMT)

diffu

`diffu` exercises the `dif()` procedure, making it act like the UNIX `diff(1)` file difference command.

Usage: diffu f1 f2 3d2 < c 7,8c6,7 < g < h −−− > i > j

Links: `dif`. (RM)

diffword

`diffword` lists all the different words in the input text. The definition of a "word" is naive. (REG)

diskpack

`diskpack` is designed to produce a list of files to fit onto diskettes. It can be adapted to other uses. This program uses a straightforward, first−fit algorithm. The options supported are:

`−s i` diskette capacity, default 360000
`−r i` space to reserve on first diskettes, default 0
`−n s` UNIX−style file name specification for files to be packed, default "*.lzh"

Requires: UNIX. Links: `options`. (REG)

duplfile

`duplfile` lists the file names that occur in more than one subdirectory and the subdirectories in which the names occur. This program should be used with caution on large directory structures. Requires: UNIX (REG)

duplproc

Use `duplproc` if you plan on posting utility procedures suitable for inclusion in someone’s Icon library directories. duplproc.icn compiles into a program that will search through every directory in your ILIBS environment variable (and/or in the directories supplied as arguments to the program). If it finds any duplicate procedure or record identifiers, it will report this on the standard output. It is important to try to use unique procedure names in programs you write, especially if you intend to link in some of the routines contained in the IPL. Checking for duplicate procedure names has been somewhat tedious in the past, and many of us (me included) must be counted as guilty

for not checking more thoroughly. Now, however, checking should be a breeze. BUGS: Duplproc thinks that differently written names for the same directory are in fact different directories. Use absolute path names, and you'll be fine.

Requires: UNIX (MS−DOS will work if all files are in MS−DOS format). (RLG)

---

envelope

`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. Usage: `envelope [options] < address(es)`

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. Links: `options`. (RF)

---

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. Notes: Not all of the farberisms contained in this program were uttered by the master himself; others have learned to emulate him. A few of the farberisms may be objectionable to some persons. "I wouldn't marry her with a twenty−foot pole." Links: `random`. (REG)

---

filecnvt

`filecnvt` copies a text file, converting line terminators.

Usage: filecnvt [−i s1] [−o s2] infile outfile

The file name "−" is taken to be standard input or output, depending on its position, although standard input/output has limited usefulness, since it translates line terminators according the system being used. The options are:

`−i s1` assume the input file has line termination for the system designated by s1. The default is "u".

`−o s2`write the output file with line terminators for the system designated by s2. The default is "u".

The designations are:

d MS−DOS ("\n\r"); also works for the Atari ST
m Macintosh ("\r")
u UNIX ("\n"); also works for the Amiga

Links: `options`. (BW)

fileprnt

`fileprnt` reads the file specified as a command−line argument and writes out a representation of each character in several forms: hexadecimal, octal, decimal, symbolic, and ASCII code. Input is from a named file rather than standard input, so that it can be opened in untranslated mode. Otherwise, on some systems, input is terminated for characters like ^Z. Since this program is comparatively slow, it is not suitable for processing very large files.

Requires: co−expressions. (REG)

filesect

`filesect` writes the section of the input file starting at a specified line number and extending a specified number of lines. The specifications are given as integer command−line arguments; the first is the starting line, the second is the number of lines. For example, `filesect 20 100 <input >output` copies 100 lines from input to output, starting at line 20 of input. If the specifications are out of range, the file written is truncated without comment. (REG)

filexref

Usage:

```
dir dir1 /b /a:d > dirlist
filexref <dirlist
```

Dir does not preface its results with the parent directory − take care! Options:

`−D`	Produce an ASCII delimited file
`−h`	Exclude hidden files
`−n`	Page Length ... must be integer >= 25

Requires: MS−DOS compatible operating system. Links: `io`, `options`. (DAG)

filtskel

`filtskel` is a generic filter skeleton in Icon. This program is not intended to be used as is −− it serves as a starting point for creation of filter programs. Command line options, file names, and tabbing are handled by the skeleton. You need only provide the filtering code. As it stands, `filtskel.icn` copies the input file(s) to standard output. Multiple files can be specified as arguments, and will be processed in sequence. The file name `"−"` represents the standard input file. If there are no arguments, standard input is processed.

Links: `options`. (RJA)

findstr

`findstr` is a utility filter to list character strings imbedded in data files (e.g. object files). Options:

`−l length`	minimum string size to be printed (default 3)
`−c chars`	a string of characters (besides the standard ASCII printable characters) to

be considered part of a string. Icon string escape sequences can be used to specify the −c option.

Multiple files can be specified as arguments, and will be processed in sequence. (RJA)

---

findtext

`findtext` retrieves multiline text from database indexed by idxtext. Each stretch of text follows a line declaring the index terms:

```
::be ::to ::by ::retrieved
Text to be retrieved
by findtext
::index ::line
Each index line begins with "::".
```

Links: `gettext`. (PLT)

---

fixpath

Usage: 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. (GMT)

---

fnctab

`fnctab` processes an MVT token file and tabulates the usage of functions. Since function usage cannot be determined completely from static analysis, the results should be viewed with this limitation in mind. (REG)

---

format

`format` is a filter program that word−wraps a range of text. A number of options are available, including full justification (see usage text, below). 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.

The `−c` option attempts to establish the form of a comment based on the first line, then does its best to deal properly with the following lines. The types of comment lines that are handled are those in which each line starts with a "comment" character string (possibly preceded by spaces). While formatting comment lines, text lines following the prototype line that don’t match the prototype but are flush with the left margin are also formatted as comments. This feature simplifies initially entering lengthy comments or making major modifications, since new text can be entered without concern for comment formatting, which will be done automatically later.

Links: `options`. (RJA)

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). (REG)

fract

`fract` produces successive rational approximations to a real number. The options supported are: `−n r` real number to be approximated, default .6180339887498948482 (see below) `−l i` limit on number of approximations, default 100 (unlikely to be reached). This program was based on a calculator algorithm posted by: Joseph D. Rudmin (duke!dukempd!jdr) Duke University Physics Dept. Aug 19, 1987.

Links: `options`. (REG, GMT)

fuzz

`fuzz` illustrates "fuzzy" string pattern matching. The result of matching s and t is a number between 0 and 1 which is based on counting matching pairs of characters in increasingly long substrings of s and t. Characters may be weighted differently, and the reverse tally may be given a negative bias. (AC)

gcomp

`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. As another example, to remove all the files in the current directory that do not match `Makefile`, `*.c`, and `*.h` the following can be used: `rm ‘gcomp Makefile *.c *.h‘` The files `.` and `..` are not included in the output, but other ‘dot files’ are.

Requires: UNIX. (WHM, REG)

genqueen

`genqueen` 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. An example of the output for n:

```
−−−−−−−−−−−−−−−
|Q| | | | | | | |
−−−−−−−−−−−−−−−
| | | | | | |Q| |
−−−−−−−−−−−−−−−
| | | | |Q| | | |
−−−−−−−−−−−−−−−
| | || | | | |Q|
−−−−−−−−−−−−−−−
| |Q| | | | | | |
−−−−−−−−−−−−−−−
| | | |Q| | | | |
−−−−−−−−−−−−−−−
| | | | | |Q| | |
−−−−−−−−−−−−−−−
| | |Q| | | | | |
```

Usage: `genqueen n`
where n is the number of rows / columns in the board. The default for n is 6. (PAB)

---

gftrace

`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. (GMT)

---

graphdem

`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. (MH)

---

grpsort

`grpsort` sorts input containing "records" defined to be groups of consecutive lines. Output is written to standard output. One or more repetitions of a demarcation line (a line beginning with the separator string) separate each input record. The first line of each record is used as the key.

If no separator string is specified on the command line, the default is the empty string. Because all input lines are trimmed of white space (blanks and tabs), empty lines are default demarcation lines. The separator string specified can be an initial substring of the string used to demarcate lines, in which case the resulting partition of the input file may be different from a partition created using the entire demarcation string.

The `−o` option sorts the input file but does not produce the sorted records. Instead it lists the keys (in sorted order) and line numbers defining the extent of the record associated with each key. The use of grpsort is illustrated by the following examples. The command `grpsort "catscats" <x >y` sorts the file `x`, whose records are separated by lines containing the string `"catscats"`, into the file `y` placing a single line of `"catscats"` between each output record. Similarly, the command `grpsort "cats" <x >y` sorts the file `x` as before but assumes that any line beginning with the string `"cats"` delimits a new record. This may or may not divide the lines of the input file into a number of records different from the previous example. In any case, a single line of `"cats"` will separate the output records. Another example is `grpsort −o <bibliography >bibkeys`, which sorts the file bibliography and produces a sorted list of the keys and the extents of the associated records in bibkeys. Each output key line is of the form: `[s−e] key` where `s` is the line number of the key line, `e` is the line number of the last line, and `key` is the actual key of the record

Links: `usage`. (TRH)

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()`.

Requires: system(), vi(1). Links: datetime, dopen, tempfile. (REG)

hebcalen,hcal4unx

This work is respectfully devoted to the authors of two books consulted with much profit: "A Guide to the Solar−Lunar Calendar" by B. Elihu Rothblatt published by the Hebrew Dept. at University of Wisconsin−Madison, and "Kiddush HaHodesh" by Rabbenu Moses ben Maimon, on whom be peace. The Jewish year harmonizes the solar and lunar cycle, using the 19−year cycle of Meton (c. 432 BCE). It corrects so that certain dates shall not fall on certain days for religious convenience. The Jewish year has six possible lengths, 353, 354, 355, 383, 384, and 385 days, according to day and time of new year lunation and position in Metonic cycle. Time figures from 6pm previous night. The lunation of year 1 is calculated to be on a Monday (our Sunday night) at ll:11:20pm. Our data table begins with a hypothetical year 0, corresponding to 3762 B.C.E. Calculations in this program are figured in the ancient Babylonian unit of halaqim "parts" of the hour = 1/1080 hour. Startup syntax is simply `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.

Links: `io`. Requires: keyboard functions, `hebcalen.dat`, `hebcalen.hlp`. (ADC, RLG)

hr

`hr` implements a horse−race game.

Links: `random`. (CT)

ibar

`ibar` replaces comment bars in Icon programs by bars 76 characters long −− the program library standard. (REG)

ibrow

`ibrow` browses Icon files for declarations. If no source file names are provided on the command line, all `*.icn` files in the current directory are browsed. The program facilitates browsing of Icon programs. It was originally written to browse the Icon Program Library, for which purpose it serves quite well. The user interface is self−explanatory −− just remember to use "?" for help if you’re confused.

Links: `colmize`. Requires: UNIX. (RJA)

icalc

`icalc` is a simple infix calculator with control structures and compound statements. It illustrates a technique that can be easily used in Icon to greatly reduce the performance cost associated with recursive−descent parsing with backtracking. Features include:

* integer and real value arithmetic
* variables
* function calls to Icon functions
* strings allowed as function arguments
* unary operators: + (absolute value), − (negation)
* assignment: :=
* binary operators: +,−,*,/,%,^,
* relational operators: =, !=, <, <=, >, >= (return 1 for true and 0 for false)
* compound statements in curly braces with semicolon separators
* if−then and if−then−else
* while−do
* limited form of multiline input

The grammar at the start of the "parser" proper provides more details. Normally, the input is processed one line at a time, in calculator fashion. However, compound statements can be continued across line boundaries. Here is a simple input:

{ a := 10; while a >= 0 do { write(a); a := a − 1 }; write("Blastoff") }

For this input execution is delayed until entire compound statement is entered. For:

```
write(pi := 3.14159)
write(sin(pi/2))
```

execution done as each line is entered. (SBW)

icalls

`icalls` processes trace output and tabulates calls of procedures (REG)

icn2c

This filter does some mundane aspects of conversion of Icon to C. It reformats comments, reformats line−continued strings, changes `:=` to `=`, reformats procedure declarations, and changes `end` to `"}"` . (RJA)

icontent

`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. The file name "−" indicates the standard input file. If there are no arguments, standard input is processed. Usage: `icontent <options> <Icon source file>...`

Options:
`−s` sort names alphabetically (default is in order of occurrence)
`−l` list in single column (default is to list in multiple columns).

(RJA)

icvt

`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. (CW, REG)

idepth

`idepth` processes trace output and reports the maximum depth of recursion. (REG)

idxtext

`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 `idxtext()`, access is faster if the indexing is done if the file is, say, over 10k (on my system the crossover point is actually about 5k).

Usage: `idxtext [-a] file1 [file2 [...]]`
where `file1`, `file2`, etc. are the names of `gettext`-format files that are to be (re-)indexed. The `-a` flag tells `idxtext` to abort if an index file already exists. Indexed files have a very simple format:

```
keyname delimiter offset [delimiter offset [etc.]]\n.
```

The first line of the index file is a pointer to the last indexed byte of the text-base file it indexes. Index files are large. Index names are not uniquely identified with their original text file. If you're worried, use the `-a` flag.

Links: `adjuncts`. See also: `gettext.icn`. (RLG, PLT)

ifilter

`ifilter` applies the operation given as a command-line argument to each line of standard input, writing out the results. For example, `ifilter reverse <foo` writes out the lines of foo reversed end-for-end. Trailing arguments can be given on the command line, as in

```
ifilter right 10 0 <foo # right(*, "10", "0")
ifilter "%" 11 <foo # * % "11"
```

Except for use with operators and (built-in) functions, this program needs to be linked with procedures that might be used with it. The following options are supported:

`-a i` argument position for strings read in; default 1
`-o i` resolution of ambiguous operator string names, 1 for unary, 2 for binary; default 2
`-l i` limit on generation, with nonpositive indicating no limitation; default 1

Links: `lists`, `options`. (REG)

igrep

`igrep` emulates UNIX egrep using the enhanced regular expressions supported by `regexp.icn`. Options supported are nearly identical to those supported by egrep (no -b: print disk block number). There is one additional option, -E, to allow 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. The following patterns are equivalent: without −E: '\bFred\b' with −E: '\\bFred\\b' (RJA)

iheader

`iheader` lists the headers of Icon programs whose file names are given on the command line. It complains if the header does not start correctly but otherwise does not check the syntax of what follows. (REG)

ihelp

ihelp displays help information. Usage: ihelp [−f helpfile] [item] [keyword ...]

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. An initial substring of the item name that differentiates it from other items is sufficient. If keyword(s) are specified, then only lines that contain all of the keywords, in any order, are displayed. The keywords do not have to correspond to whole words in the help text; only to text fragments. All item name and keyword matches are case independent. The help file name is taken from environment variable "HELPFILE". If HELPFILE is not in the environment, file "help" in the current directory is used. A help file name specified in the `−f` option overrides. The help files are formatted as follows:

```
default text lines
−
one
item "one" text lines
−
two
item "two" text lines
...
```

Sections are separated by lines containing a single "−". Item names are the first line following a separator line.

Links: `options`. (RJA)

iidecode, iiencode

These are Icon ports of the UNIX/C uudecode/uuencode utilities, based on freely distributable BSD code. The functional changes to the program are: (1) file modes are always encoded with 0644 permissions, and (2) a command−line switch was added for xxencoded files (similar to uuencoded files, but capable of passing unscathed through non−ASCII EBCDIC sites).

Usage: iidecode [infile] [−x]

`iidecode` is compatible with that of the UNIX uudecode command. A first (optional) argument gives the name the file to be decoded. If this is omitted, `iidecode` just uses the standard input. The `−x` switch (peculiar to `iidecode`) forces use of the xxdecoding algorithm. If you try to decode an xxencoded file without specifying `−x` on the

command line, `iidecode` will try to forge ahead anyway. If it thinks you've made a mistake, `iidecode` will inform you after the decode is finished.

`iiencode`'s usage is compatible with that of the UNIX uuencode command. A first (optional) argument gives the name the file to be encoded. If this is omitted, `iiencode` just uses the standard input. The second argument specifies the name the encoded file should be given when it is ultimately decoded. Extensions to the base uuencode command options include `−x` and `−o`. An `−x` tells `iiencode` to use xxencode (rather than uuencode) format. Option `−o` causes the following argument to be used as the file `iiencode` is to write its output to (the default is `&output`). On systems with newline translation (e.g. MS−DOS), the `−o` argument should always be used.

Usage: iiencode [infile] [−x] remote−filename [−o output−filename]

(RLG, FJL)

---

ilnkxref

`ilnkxref` is a utility to create cross−reference of library files used in Icon programs (i.e., those files named in "link" declarations).

Usage: ilnkxref [−options] <icon source file>...

`−p` sort by "popularity"
`−v` report progress information

Requires: UNIX. Links: `wrap`, `options`, `isort`. (RJA)

---

ilump

`ilump` copies one or more Icon source files, incorporating recursively the source code for files named by "link" directives. This produces a standalone source program in one file, which is useful with certain profiling and visualization tools. Searching for linked source files is similar to the action of Iconc under UNIX. If a linked file is not found in the current directory, directories specified by the LPATH environment variable are tried. (GMT)

---

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`. This is problematic and corrupted or fake files can easily fool it. Furthermore, examples of some image file types were not available for testing. The types presently recognized are:

Value returned	Image file type
ps	PostScript document
cgm text	Computer Graphics Metafil, text
cgm binary	Computer Graphics Metafil, binary
cgm char	Computer Graphics Metafil, character
sundraw	SunDraw document

ras	UNIX raster image
iris	Iris image
rle	UNIX RLE image
pbm	PBM image
pgm	PGM image
ppm	PPM image
xwd	X Window dump
gif	Compuserv GIF image
bmp	BMP image
xmp	XMP image
xpm	XPM image
pcx	PCX image
tiff	TIFF image
iff	IFF/ILBM image
?	unknown type

Links: `bincvt`. (REG)

---

ineeds

`ineeds` determines Icon modules required by an Icon module. It expects environment variable LPATH to be set properly as for the Icon Compiler. (RJA)

---

inter

`inter` lists lines common to two files. (REG)

---

interpe, interpp

`interpe` is a crude but effective interpreter for Icon expressions. Each line entered from standard input must be an Icon expression. The expression is wrapped with a main procedure, and written to a pipe that compiles and executes the resulting program. If the expression is a generator, all its results are produced. If the command−line option −e is given, the expression is echoed. This technique is, of course, inefficient and may be painfully slow except on the fastest platforms. However, the technique is completely general and as correct as Icon itself.

`interpp` is kind of like an interactive version of BASIC in that Icon expressions are entered with line numbers and you can resequence them list them etc. and execute all the lines entered. There is no editor built in. You have to retype a line to change it. Documentation is lacking but there is a "?" help command that lists all the other commands.

(REG, JN)

---

ipatch

Usage: ipatch file path

`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.

Requires: Unix. (GMT)

---

ipldoc

`ipldoc` collects selected information from documentation headers for Icon procedure files named on the command line. The following options are supported:

`-s` skip file headers
`-f` sort procedure list by file; default sort by procedure name

Links: `options`, `sort`. (REG)

---

iplindex

`iplindex` produces an indexed listing of the Icon Program Library. The following options are supported:

`-k i` width keyword field, default 16
`-p i` width of field for program name, default 12

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.

Links: `options`. (REG, SBW)

---

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. It reads from standard input and writes to standard output. 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 characters whose "print width" is less than one may not be handled properly. The following options are supported:

`-c i` column at which keywords are aligned, default 30
`-h i` width of identifying column at left, default 20

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.

Links: `options`. (SBW, REG)

---

iplweb

Usage: iplweb [−ipl source] [dest]

`iplweb` generates web pages from IPL header comments. It uses an environment variable IPL which is a path to the Icon Program Library as a default if `-ipl` is not

specified, `dest` is the current directory if not specified. `iplweb` generates an HTML subdirectory in `dest` and makes an index to `gprogs`, `gprocs`, `procs`, and `progs` directories under HTML. In each of these directories is an `.html` file for each of the `.icn` files in the referenced directory. An index to all of these files is also generated. Each of the `.html` files contains the IPL standard comment header info inside. (JK)

---

iprint

`iprint` is a program to print Icon programs. If a program is written in a consistent style, this program attempts to keep whole procedures on the same page. The default is to identify the end of a print group (i.e. a procedure) by looking for the string "end" at the beginning of a line. Through the −g option, alternative strings can be used to signal end of a group. Using "end" as the group delimiter (inclusive), comments and declarations prior to the procedure are grouped with the procedure. Specifying a null group delimiter string (`−g ''`) suppresses grouping. Page creases are skipped over, and form−feeds (^L) imbedded in the file are handled properly. (Form−feeds are treated as spaces by many C compilers, and signal page ejects in a listing). Page headings (file name, date, time, page number) are normally printed unless suppressed by the `−h` option. Options:

`−n` number lines
`−pN` page length: number of lines per page (default: 60 lines)
`−tN` tab stop spacing (default: 8)
`−h` suppress page headings.
`−l` add three lines at top of each page for laser printer.
`−gS` end of group string (default: "end").
`−cS` start of comment string (default: "#").
`−xS` end of comment string (default: none).
`−i` ignore FF at start of line.

Any number of file names specified will be printed, each starting on a new page. For example, to print C source files such as the Icon source code, use the following options:

```
iprint −g ' }' −c '/*' −x '*/' file ...
```

Control lines are special character strings that occur at the beginnings of lines that signal special action. Control lines begin with the start of comment string (see options). The control lines currently recognized are:

`<comment string>eject` −− page eject (line containing "eject" does not print).
`<comment string>title` −− define a title line to print at top of each page. Title text is separated from the `<comment string>title` control string by one space and is terminated by `<end of comment string>` or end of line, whichever comes first.
`<comment string>subtitle` −− define a subtitle line to print at top of each page. Format is parallel to the "title" control line, above.

If a page ejection is forced by maximum lines per page being exceeded (rather than intentional eject via control line, form feed, or grouping), printing of blank lines at the top of the new page is suppressed. Line numbers will still be printed correctly. (RJA)

ipsort

`ipsort` reads an Icon program and writes an equivalent program with the procedures sorted alphabetically. Global, link, and record declarations come first in the order they appear in the original program. The main procedure comes next followed by the remaining procedures in alphabetical order. Comments and white space between declarations are attached to the next following declaration. Limitations: This program only recognizes declarations that start at the beginning of a line. Comments and interline white space between declarations may not come out as intended. One option is accepted: −v preserve VIB section at end

Links: `options`. (REG)

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 the `−g` option 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. Comments and white space between declarations are attached to the next following declaration. Notes: The program only recognizes declarations that start at the beginning of lines. Comments and interline white space between declarations may not come out as intended. If the −g option is not specified, any global, link, or record declarations that follow the last procedure are discarded.

Links: `options`. (REG)

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−alphanumerics. 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 format 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. These options expand what is considered:

`−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.

Links: `options`. (AJA)

isrcline

`isrcline` counts the number of lines in an Icon program that actually contain code, as opposed to being comments or blank lines.

Links: `numbers`. (REG)

istrip

`istrip` removes comments from an Icon program. It also removes empty lines and leading white space.

Links: `stripcom`. (REG)

itab

`itab` entabs an Icon program, leaving quoted strings alone.

Usage: itab [options] [source−program...]

Options:
`−i` Input tab spacing (default 8)
`−o` Output tab spacing (default 8).

`itab` observes Icon Programming Language conventions for escapes and continuations in string constants. If no source−program names are given, standard input is "itabbed" to standard output.

Links: `options`, `io`. (RJA)

itags

`itags` creates a tags file for an Icon program.

Usage: itags [−aBFtvwx] [−f tagsfile] file...

The options are:

`−a` append output to an existing tags file.
`−B` use backward searching patterns (?...?).
`−F` use forward searching patterns (/.../) (default).
`−x` produce a list of 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 is a simple index that 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.

Links: isort, io, options. (RJA)

---

itrbksum

itrbksum summarizes traceback information produced on error termination by filtering out the bulk of the procedure traceback information. Expect various options in future versions. (REG)

---

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.

Links: itrcline, options. (REG)

---

itrcsum

itrcsum provides a summary of Icon trace output.

Links: itrcline, numbers. (REG)

---

iundecl

iundecl 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 will be processed in sequence. The file name "−" represents the standard input file. If there are no arguments, standard input is processed. The program works only if procedures are formatted such that the keywords "procedure" and "end" are the first words on their respective lines. Only for UNIX, since the "p" (pipe) option of open() is used.

Requires: UNIX. Links: io. (RJA, REG)

---

iwriter

iwriter reads standard input and produces Icon expressions, which when compiled and executed, write out the original input. This is handy for incorporating, for example, message text in Icon programs. Or even for writing Icon programs that write Icon programs that ... (REG)

---

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. It is derived from fillup.icn, which has a bewildering array of options to make it applicable to real−world problems. In contrast, knapsack is merely a demonstration of the underlying algorithm.

Knapsack may be tested conveniently by piping to it the output of `randi`, a trivial program, like this:

```
randi 100 10 | knapsack 250
```

You may pick a different capacity, of course; this one just happens to produce a result quite quickly, as you might expect. (AVH)

---

krieg

Kriegspiel (German for "war game") implements a monitor and, if desired, an automatic opponent for 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. Thus Kriegspiel combines the intricacies and flavor of chess with additional elements of uncertainty, psychology, subterfuge, etc., which characterize games of imperfect information such as bridge or poker. To start the game, the "White" player makes a move on his board. If the move is legal, the monitor plays it on his board and invites "Black" to make his response. If a move attempt is illegal (because it leaves the king in check or tries to move through an enemy piece, etc.), the monitor announces that fact to both players and the moving player must try again until he finds a legal move. The game continues until it ends by checkmate, draw, or agreement by the players. The monitor keeps a record of the moves so that the players can play the game over at its conclusion and see what actually happened, which is often quite amusing. (DJS)

---

kross

`kross` accepts pairs of strings on successive lines. It diagrams all the intersections of the two strings in a common character. (REG)

---

kwic, kwicprep

`kwic` is a simple keyword−in−context (KWIC) program. It reads from standard input and writes to standard output. 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 used for creating keyword−in−context listings of the Icon program library. (SBW, REG)

---

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. The following options are available:

`-c n` Print n copies of each label.
`-s s` Select only those labels whose headers contain a character in s.
`-t` Format for curved tape labels (the default is rectangular mailing labels).
`-w n` Limit line width to n characters. The default width is 40.
`-l n` Limit the number of printed lines per label to n. The default is 8.
`-d n` Limit the depth of the label to n. The default is 9 for rectangular labels and 12 for tape labels (`-t`).

Options are processed from left to right. If the number of printed lines is set to a value that exceeds the depth of the label, the depth is set to the number of lines. If the depth is set to a value that is less than the number of printed lines, the number of printed lines is set to the depth. Note that the order in which these options are specified may affect the results. Printing Labels: Label forms should be used with a pin−feed platen. For mailing labels, the carriage should be adjusted so that the first character is printed at the leftmost position on the label and so that the first line of the output is printed on the topmost line of the label. For curved tape labels, some experimentation may be required to get the text positioned properly. Diagnostics: If the limits on line width or the number of lines per label are exceeded, a label with an error message is written to standard error output.

Links: options, io. See also: address.doc, adl*.icn, zipsort.icn. (REG)

---

lam

`lam` laminates files named on the command line onto the standard output, producing a concatenation of corresponding lines from each file named. If the files are different lengths, empty lines are substituted for missing lines in the shorter files. A command line argument of the form `-s` causes the 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 the command lam file1 −: file2. File names and strings may appear in any order in the argument list. If − is given for a file name, standard input is read at that point. If a file is named more than once, each of its lines will be duplicated on the output line, except that 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 the command lam file1 "− " file1 replicates each line from file1.

Links: `usage`. (TRH)

---

latexidx

`latexidx` processes LaTeX idx files. Input: A latex .idx file containing the \indexentry lines. Output: \item lines sorted in order by entry value, with page references put into sorted order. (DSC)

---

lc

`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. (REG)

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`. Links: `options`. (REG)

lineseq

`lineseq` reads values on separate lines and strings them together on a single line. The default separator is a blank; other separating strings can be specified by the `−s` option

Links: `options`. (REG)

lisp

`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 Laurent Siklossy's "Let's Talk LISP" (Prentice−Hall, 1976, ISBN 0−13−532762−8). 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). (SBW, PLT)

literat

`literat` is a database system to manage information concerning literature. The program uses standard files `literat.fil`, `literat2.fil` and `adress.fil` to store its data on the disk. It has a predefined structure of the items and predefined field labels to make it easy to use and to cut down the source code length.

Requires: ANSI terminal support, e.g. `ansi.sys`. (MH)

loadmap

`loadmap` produces a formatted listing of selected symbol classes from a compiled file. The listing is by class, and gives the name, starting address, and length of the region associated with each symbol. The options are:

`−a` Display the absolute symbols.
`−b` Display the BSS segment symbols.
`−c` Display the common segment symbols.
`−d` Display the data segment symbols.
`−t` Display the text segment symbols.
`−u` Display the undefined symbols.

If no options are specified, `-t` is assumed. 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.

Requires: UNIX. (SBW)

---

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. (REG)

---

lower

`lower` maps the names of all files in the current directory to lowercase.

Requires: UNIX. (REG)

---

makepuzz

`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. Usage:

makepuzz [−f input−file] [−o output−file] [−h puzzle−height] [−w puzzle−width]
[−t seconds−to−keep−trying] [−r maximum−number−of−rejects][−s] [−d]

where input−file is a file containing words, one to a line (defaults to `&input`), and output−file is the file you would like the puzzle written to (defaults to `&output`). Puzzle−height and 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, by putting random letters in all the blank spaces. The `−t` tells the computer when to give up, and construct the puzzle (letting you know if any words didn't make it in). Defaults to 60 (i.e. one minute). The `−r` argument tells `makepuzz` to run until it arrives at a solution with number−of−rejects or fewer un−inserted words. `−d` turns on certain diagnostic messages. Most of these options can safely be ignored. Just type something like "`makepuzz −f wordlist`", where `wordlist` is a file containing about sixty words, one word to a line. Out will pop a "word−find" puzzle. Once you get the hang of what is going on, try out the various options. The algorithm used here is a combination of random insertions and mindless, brute−force iterations through possible insertion points and insertion directions. If you don't like `makepuzz`'s performance on one run, run it again. If your puzzle is large, try increasing the timeout value (see `−t` above).

Links: options, random, colmize. (RLG)

---

missile

`missile` is a cheap attempt at a Missile Command game. It runs on systems that support the `delay()` procedure, and uses ANSI escape sequences for screen output. Also 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.

Links: `random`. Requires: ANSI terminal, e.g. `ansi.sys`. (CT)

miu

`miu` generates strings from the MIU string system. The number of generations is determined by the command−line argument. The default is 7. Reference: Godel, Escher, and Bach: an Eternal Golden Braid, Douglas R. Hofstadter, Basic Books, 1979. pp. 33−36. (CAC, REG)

mkpasswd

`mkpasswd` creates a list of randomly generated passwords. Passwords consist of eight random characters [A−Z][0−9]. Number of passwords to generate is given as the first argument; default 1. genpasswd: generate and return an 8−character password (JK)

monkeys

The old monkeys at the typewriters anecdote... `monkeys` uses ngram analysis to randomly generate text in the same 'style' as the input text. The arguments are:

`−s` show the input text
`−n` n use n as the ngram size (default:3)
`−l n` output at about n lines (default:10)
`−r n` set random number seed to n

Links: `options`. (SBW, REG, AB)

morse

If `morse` is invoked without arguments, a Morse code table is printed. If words are entered as arguments, the Morse code conversion is printed in dots and dashes. If the first character of the first argument is a dot or dash, the arguments are takes as Morse code and converted to a string.

Links: `colmize`. (REG, RJA)

mr

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. Usage:

mr [recipient] [−u user] [−f spool]

The program is oriented towards UNIX Internet mail processing.

Links: iolib, options, tempname. (RF)

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 supported 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()`.

Requires: system(), vi(1). Links: basename, datetime, io, options. (REG)

---

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. (ADC)

---

nim

The game of `nim` focuses on a pile of 15 sticks. Each player can select 1, 2, or 3 sticks from the sticks remaining in the pile when it's 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 in here. The first (default) version uses an algorithm to make its moves. It will never lose if it gets the first turn. The second version tries to learn from each game. You'll have to play a few games before it will get very smart but after a while it will also never lose if it gets the first turn. This is assuming of course that you know how to play. Since the learning version learns from the person it plays against, if you're lousy the game will be too. To invoke the learning version, just pass any argument to the program. If you want to see how the program learns, you can use the string "show" as the argument and the program's current game memory will be displayed after each game. If you invoke the game with the string save as an argument a file called ".nimdump" will be created in the current directory with a dump of the program's game memory when you quit. The next time the game is played in learn mode it will initialize its game memory from the dump. You can invoke this program with more than one argument so show and save can be used at the same time.

Links: `random`. (JN)

---

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. Links: `datetime`, `options`. (REG)

pack

`pack` takes a list of file names on the command line and packages the files 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`. (REG)

paginate

`paginate` processes a document text file, inserting form feeds at appropriate places. (PA)

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. As it stands, there is no way to provide other arguments. That's easy to remedy. Except for use with (built−in) functions, this program needs to be linked with procedures that might be used with it. (REG)

parens

`parens` produces parenthesis−balanced strings in which the parentheses are randomly distributed. The following options are available:

`−b n` Bound the length of the strings to n left and right parentheses each. Default: 10.
`−n n` Produce n strings. Default :10.
`−l s` Use the string s for the left parenthesis. Default: `"("` .
`−r s` Use the string s for the right parenthesis. Default: `")"` .
`−v` Randomly vary the length of the strings between 0 and the bound. In the absence of this option, all strings are the exactly as long as the specified bound.

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 some way of generating other text among the parentheses. In addition to the intended use of the program, it can produce a variety of interesting patterns, depending on the strings specified by `−l` and `−r`.

Links: `options`, `random`. (REG)

pargen

`paregen` reads a context−free BNF grammar and produces an Icon program that is a parser for the corresponding language. Nonterminal symbols are enclosed in angular brackets. Vertical bars separate alternatives. All other characters are considered to be terminal symbols. The nonterminal symbol on the first line is taken to be the goal. An example is:

```
<expression>::=<term>|<term>+<expression>
<term>::=<element>|<element>*<term>
<element>::=x|y|z|{<expression>}
```

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. Characters in nonterminal names are limited to letters, digits, and underscores. An underscore is appended to the parsing procedure name to avoid possible collisions with Icon function names. Lines beginning with an = are passed through unchanged. This allows Icon declarations to be placed in the parser. Lines beginning with a # are considered to be 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. Otherwise the main procedure in `recog` is used. Limitations: Left recursion in the grammar may cause the parser to loop. There is no check that all nonterminal symbols that are referenced are defined or that there may be duplicate definitions. Output links `recog`, `matchlib`. See also: `recog.icn`, `matchlib.icn`, and `parscond.icn` in the IPL source code. (REG)

---

parse, parsex

`parse` parses simple statements. It provides an interesting example of the use of co−expressions. (KW)

`parsex` is another expression parser, adapted from C code written by Allen I. Holub published in the Feb 1987 issue of Dr. Dobb's Journal. 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. (CW)

---

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` for the hard work. 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. Links: `options`, `patch`. (RM)

---

pdecomp

`pdecomp` lists the prime factors of integers given in standard input.

Links: `factors`. (REG)

---

polydemo

`polydemo` is an example for the use of the `polystuf` library module. The user is given a number of options that allow the creation, output, deletion, or operations on 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

Links: `polystuf`. (EE)

---

post

`post` posts a news article to Usenet. Given an optional argument of the name of a file containing a news article, or an argument of `"−"` and a news article via standard input, `post` creates a follow−up article, with an attribution and quoted text. The newsgroups, subject, distribution, follow−up, and quote−prefix can optionally be specified on the command line. Usage:

post [options] [article | −] −n newsgroups −s subject −d distribution
    −f followup−to −p quote−prefix (default ‘ > ’)

On systems posting via `inews`, `post` validates newsgroups and distributions in the ‘active’ and ‘distributions’ files in the news library directory. Bugs: Newsgroup validation assumes the ‘active’ file is sorted. Non−UNIX sites need hardcoded system information.

Links: `options`. (RF)

---

press

Besides being a useful file archiving utility, `press` can be used to experiment with the LZW compression process, as it contains extensive tracing facilities that illustrate the process in detail. Compression can be turned off if faster archiving is desired. The LZW compression procedures in this program are general purpose and suitable for reuse in other programs.

Links: options, colmize, wildcard. (RJA)

---

procprep

`procprep` is used to produce the data needed to index the "#:" comments on procedure declarations that is needed to produces a permuted index to procedures. (REG)

---

procwrap

`procwrap` takes procedure names from standard input and writes minimal procedure declarations for them. For example, the input line wrapper produces:

```
procedure wrapper()
end
```

This program is useful when you have a lot of procedures to write. (REG)

psrsplit

psrsplit separates psrecord.icn output pages. Usage: psrsplit file

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. The algorithm is frugal of memory and file descriptors at the expense of reading the input file multiple times. 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. (GMT)

puzz

`puzz` creates word search puzzles. (CT)

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 clear the accumulated expressions.
:every generate all the results from the expression; otherwise, at most one is produced.
:exit terminate the session
:quit terminate the session
:list list the accumulated expressions.
:type toggle switch that displays the type of the result; the program starts with this switch on.

"qei" is derived from the Latin "quod erat inveniendum" −− "which was to be found out".

Requires: co−expressions and `system()`. (WHM, REG)

qt

`qt` writes out the time in English. Usage: `qt [−a]`

If `−a` is present, only the time is printed (for use in scripts), e.g.: just after a quarter to three otherwise, the time is printed as a sentence: It’s just after a quarter to three.

Links: `datetime`. (RJA)

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. A positive integer can be given as a command line argument to specify the number of queens. For example, `queens −n8` displays the solutions for 8 queens on

an 8−by−8 chessboard. The default value in the absence of an argument is 6. This program is worth reading for its programming techniques.

Links: `options`. (SBW)

---

ranstars

`ranstars` displays a random field of "stars" on an ANSI terminal. It displays stars 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. It is originally due to Steve Wampler. The options are:

`−m n` maximum number of stars, default 10.
`−t n` length of steady−state time before stars are extinguished, default 50.
`−s c` the character used for "stars", default *. Only the first character in `c` is used.

Requires: co−expressions, ANSI terminal. Links: `ansi`, `options`, `random`. (REG)

---

recgen

`recgen` reads a context−free BNF grammar and produces an Icon program that is a recognizer for the corresponding language. Nonterminal symbols are enclosed in angular brackets. Vertical bars separate alternatives. All other characters are considered to be terminal symbols. The nonterminal symbol on the first line is taken to be the goal. An example is:

```
<expression>::=<term>|<term>+<expression>
<term>::=<element>|<element>*<term>
<element>::=x|y|z|(<expression>)
```

Characters in nonterminal names are limited to letters and underscores. An underscore is appended for the recognizing procedure name to avoid possible collisions with Icon function names. Lines beginning with an = are passed through unchanged. This allows Icon code to be placed in the recognizer. Limitations: Left recursion in the grammar may cause the recognizer to loop. There is no check that all nonterminal symbols that are referenced are defined or for duplicate definitions. Reference: The Icon Programming Language, Second Edition, Ralph E. and Madge T. Griswold, Prentice−Hall, 1990. pp. 180−187. See also: `pargen.icn`. (REG)

---

reply

`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 ‘> ’. Usage: `reply [prefix] <` *`news−article`* or *`mail−item`*

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. (RF)

repro

`repro` is the shortest known self−reproducing Icon program. (KW)

revsort

`revsort` sorts strings with characters in reverse order. (REG)

roffcmds

`roffcmds` processes standard input and writes a tabulation of nroff/troff commands and defined strings to standard output. Limitations: This 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. Reference: Nroff/Troff User's Manual, Joseph F. Ossana, Bell Laboratories, Murray Hill, New Jersey. October 11, 1976. (REG)

rsg

`rsg` generates randomly selected strings ("sentences") from a grammar specified by the user. Grammars are basically context−free and resemble BNF in form, although there are a number of extensions. The program works interactively, allowing the user to build, test, modify, and save grammars. Input to `rsg` consists of various kinds of 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. In addition, any line beginning with # is considered to be a comment, while any line beginning with = causes the rest of that line to be used subsequently 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. Productions: Examples of productions are:

```
<expr>::=<term>|<term>+<expr>
<term>::=<elem>|<elem>*<term>
<elem>::=x|y|z|(<expr>)
```

Productions may occur in any order. Specifying a new production for it changes the definition for a nonterminal symbol. There are a number of special devices to facilitate the definition of grammars, including eight predefined, built−in nonterminal symbols: symbol definition <lb> < <rb> > <vb> | <nl> newline <> empty string <&lcase> any single lowercase letter <&ucase> any single uppercase letter <&digit> any single digit. 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

Generation Specifications:

A generation specification consists of a nonterminal symbol followed by a nonnegative integer. An example is <expr>10 which specifies the generation of 10 <expr>s. If the integer is omitted, it is assumed to be 1. Generated sentences are written to standard

output. Grammar Output Specifications: A grammar output specification consists of a nonterminal symbol, followed by −>, followed by a file name. Such a specification causes the current definition of the nonterminal symbol to be written to the given file. If the file is omitted, standard output is assumed. If the nonterminal symbol is omitted, the entire grammar is written out. Thus, −> causes the entire grammar to be written to standard output.

Source Specifications:

A source specification consists of @ followed by a file name. Subsequent input is read from that file. When an end of file is encountered, input reverts to the previous file. Input files can be nested.

Options:

The following options are available: `−s n` Set the seed for random generation to n. `−r` In the absence of `−s`, set the seed to 0 for repeatable results. Otherwise the seed is set to a different value for each run (as far as this is possible). `−r` is equivalent to `−s 0`. `−l n` Terminate generation if the number of symbols remaining to be processed exceeds n. The default is limit is 1000. `−t` Trace the generation of sentences. Trace output goes to standard error output.

Diagnostics:

Syntactically erroneous input lines are noted but are otherwise ignored. Specifications for a file that cannot be opened are noted and treated as erroneous. If an undefined nonterminal symbol is encountered during generation, an error message that identifies the undefined symbol is produced, followed by the partial sentence generated to that point. Exceeding the limit of symbols remaining to be generated as specified by the −l option is handled similarly. Caveats: Generation may fail to terminate because of a loop in the rewriting rules or, more seriously, because of the progressive accumulation of nonterminal symbols. The latter problem can be identified by using the −t option and controlled by using the −l option. Duplicating alternatives that lead to fewer rather than more nonterminal symbols often can circumvent the problem. For example, changing

```
<term>::=<elem>|<elem>*<term>
```

to

```
<term>::=<elem>|<elem>|<elem>*<term>
```

increases the probability of selecting `<elem>` from 1/2 to 2/3.

Links: `options`, `random`. (REG)

---

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. (RJA)

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.

Links: `random`. (CT)

setmerge

`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. Usage: `setmerge file [[op] file]...`

Operations:

+ add contents to set
− subtract contents from set
* intersect contents with set

Note that operators must be separate command options, and that some shells my require some of them to be quoted.

Example 1: combine files, sorting and eliminating duplicates:

```
setmerge file1 + file2 + file3 + file4
```

Example 2: print lines common to three files

```
setmerge file1 '*' file2 '*' file3
```

Example 3: print lines in file1 or file2 but not in file3

```
setmerge file1 + file2 − file3
```

(GMT)

shar

`shar` creates Bourne shell archive of text files. Usage: `shar text_file...` (RJA)

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. (REG)

shuffile

`shuffile` writes a version of the input file with the lines shuffled. For example, the result of shuffling

```
          On the Future!−how it tells
          Of the rapture that impells
        To the swinging and the ringing
          Of the bells, bells, bells−
      Of the bells, bells, bells, bells,
```

```
                Bells, bells, bells–
   To the rhyming and the chiming of the bells!
```

is

```
   To the rhyming and the chiming of the bells!
         To the swinging and the ringing
                 Bells, bells, bells–
          Of the bells, bells, bells–
          On the Future!–how it tells
       Of the bells, bells, bells, bells,
          Of the rapture that impells
```

The following options are supported:

`–s i` Set random seed to `i`; default 0
`–r` Set random seed using `randomize()`; overrides `–s`

This program stores the input file in memory and shuffles pointers to the lines; there must be enough memory available to store the entire file.

Links: `options`, `random`. (REG)

---

sing

`sing` is an Icon adaptation of a SNOBOL program by Mike Shapiro in the book The SNOBOL4 Programming Language. 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". Reference: "The SNOBOL 4 Programming Language" by Griswold, Poage, and Polonsky, 2nd ed. Englewood Cliffs, N.J. Prentice–Hall, Inc. 1971. (FJL)

---

snake

While away the idle moments watching the snake eat blank squares on your screen. Usage: `snake [character]` where `character` represents a single character to be used in drawing the snake. The default is an `"o"`. In order to run `snake`, your terminal must have cursor movement capability, and must be able to do reverse video. Bugs: Most magic cookie terminals just won’t work. Terminal really needs reverse video (it will work without, but won’t look as cute).

Links: `iolib`, `iscreen`, `random`. (RLG)

---

solit

`solit` was inspired by a solitaire game that was written by Allyn Wade and copyrighted by him in 1985. His game was designed for the IBM PC/XT/PCjr with a color or monochrome monitor. I didn’t follow his design exactly because I didn’t want to restrict myself to a specific machine. This program has the correct escape sequences programmed into it to handle several common terminals and PC’s. It’s commented well enough that most people can modify the source to work for their hardware.

This program is about as totally commented as you can get so the logic behind the autopilot is fairly easy to understand and modify. It's up to you to make the auto pilot smarter. Note: The command−line argument, which defaults to support for the VT100, determines the screen driver. For MS−DOS computers, the ANSI.SYS driver is needed.

Requires: keyboard functions. (JN, PLT, REG)

---

sortname

`sortname` sorts a list of person's names by the last names. (REG)

---

splitlit

`splitlit` creates a string literal with continuations in case it's too long. Options:

`−w i` width of piece on line, default 50
`−i i` indent, default 3

Links: `options`. (REG)

---

streamer

`streamer` outputs one long line obtained by concatenating the lines of the input file. The supported options are:

`−l i` stop when line reaches or exceeds i; default no limit
`−s s` insert s after each line; default no separator

Separators are counted in the length limit.

Links: `options`. (REG)

---

strimlen

`strimlen` is a filter that reads images of Icon strings from standard input and writes the lengths of the strings to standard output.

Links: `ivalue`. (REG)

---

strpsgml

`strpsgml` strips or performs simple translation on SGML <>−style tags.

Usage: strpsgml [−f translation−file] [left−delimiter [right−delimiter]]

The default left−delimiter is <, the default right delimiter is >. If no translation file is specified, the program acts as a stripper, simply removing material between the delimiters. `strpsgml` takes its input from standard input and writes to standard output. 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. There is not currently any way of specifying a completion field without an initialization field. Do not specify delimiters as part of code. Note that, if you are translating SGML code into font change or escape sequences, you may get unexpected results. This isn't `strpsgml`'s fault. It's just a matter of how your terminal or WP operate. Some need to be "reminded" at the beginning of each line what mode or font is being used. If you want to put a greater−than or less−than sign into your text, put a backslash before it. This will effectively "escape" the special meaning of those symbols. It is possible to change the default delimiters, but the option has not been thoroughly tested.

Links: scana, stripunb, readtbl. (RLG)

---

tablc

`tablc` tabulates characters and lists each character and the number of times it occurs. Characters are written using Icon's escape conventions. Line termination characters and other control characters are included in the tabulation. The following options are available:

`−a` Write the summary in alphabetical order of the characters. This is the default.
`−n` Write the summary in numerical order of the counts.
`−u` Write only the characters that occur just once.

Links: `options`. (REG)

---

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. The following options are available:

`−a` Write the summary in alphabetical order of the words. This is the default.
`−I` Ignore case distinctions among letters; uppercase letters are mapped to corresponding lowercase letters on input. The default is to maintain case distinctions.
`−n` Write the summary in numerical order of the counts.
`−l n` Tabulate only words longer than n characters. The default is to tabulate all words.
`−u` Write only the words that occur just once.

Links: `options`, `usage`. (REG)

---

textcnt

`textcnt` tabulates the number of characters, "words", and lines in standard input and gives the maximum and minimum line length. (REG)

---

textcvt

`textcvt` converts text file(s) among various platforms' formats. The supported text file types are UNIX, MS−DOS, and Macintosh. A universal input text reading algorithm

is used, so only the output file format must be specified. 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. If the conversion is interrupted, the temporary file might still remain as <original name>.temp (or, for MS−DOS, <original name root>.tmp.

Links: `io`, `options`. (RJA)

---

toktab

`toktab` reads the token files given on the command line and summarizes them in a single file. The supported options are:

`−n` sort tokens by category in decreasing *numerical* order; default is alphabetical
`−l i` limit output in any category to i items; default no limit

Links: `options`, `showtbl`. (REG)

---

trim

`trim` copies lines from standard input to standard output, truncating the lines at n characters and removing any trailing blanks. The default value for n is 80. For example,

```
trim 70 <grade.txt >grade.fix
```

copies `grade.txt` to `grade.fix`, with lines longer than 70 characters truncated to 70 characters and the trailing blanks removed from all lines. The `−f` option causes all lines to be n characters long by adding blanks to short lines; otherwise, short lines are left as is.

Links: `options`. (REG)

---

ttt

`ttt` plays the game of tic−tac−toe.

Links: `random`. (CT)

---

turing

`turing` simulates the operation of an n−state Turing machine, tracing all actions. 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:

```
sss. wdnnn wdnnn
```

`sss` is the state number in decimal. The `wdnnn` fields specify the action to be taken on reading a 0 or 1 respectively: `w` is the digit to write (0 or 1), `d` is the direction to move (L/l/R/r, or H/h to halt), `nnn` is the next state number (0 if halting).

Sample input file:

```
1. 1r2 1l3
2. 1l1 1r2
3. 1l2 1h0
```

One line is written for each cycle giving the cycle number, current state, and an image of that portion of the tape that has been visited so far. The current position is indicated by reverse video (using ANSI terminal escape sequences). Input errors are reported to standard error output and inhibit execution. Bugs: Transitions to nonexistent states are not detected. Reverse video should be parameterizable or at least optional. There is no way to limit the number of cycles. Infinite loops are not detected. (Left as an exercise...`:-)` Reference: Scientific American, August 1984, pp. 19−23. A. K. Dewdney's discussion of "busy beaver" Turing machines in his "Computer Recreations" column motivated this program. The sample above is the three−state busy beaver.

Links: `options`. (GMT)

---

unique

`unique` filters out (deletes) identical adjacent lines in a file. (AVH, RJA)

---

unpack

`unpack` unpackages files produced by `pack.icn`. See that program for information about limitations. (REG)

---

upper

`upper` maps the names of all files in the current directory to uppercase.

Requires: UNIX. (REG)

---

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 `verse.dat` by default. See that file for examples of form.

Links: `random`. (CT)

---

versum

`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. The supported options are:

`−s i` The seed for the sequence, default 196
`−f s` Name of file to extend, no default
`−F s` Name of file, default <i>.vsq, where <i> is the seed of the sequence
`−t i` The number of steps to carry the sequence out to, default unlimited
`−m i` Stop when value equals or exceeds m; default no limit

If both `−f` and `−F` are given, `−f` overrides.

Links: `options`. (REG)

---

vnq

`vnq` displays solutions to the n−queens problem.

Links: `options`. (SBW)

---

webimage

`webimage` takes the names of image files on the command line and writes a Web page that embeds each image. The following options are supported:

`−a s` alignment, default "bottom"
`−t s` title for page; default "untitled"
`−n` include file names; default no names

Requires: Version 9 graphics. Links: `options`, `wopen`. (REG)

---

what

`what` writes all strings beginning with "@" followed by "(#)" and ending with null, newline, quotes, greater−than or backslash. Follows UNIX `what(1)` conventions.

Links: `basename.icn`. (PLT)

---

when

`when` was developed for ULTRIX 4.3 rev 44. Maybe it will work on some other versions of UNIX too. `when` is like a date based `ls` command. UNIX `find` can do the same things, but `find` is a bit arcane, so `when` provides a simpler alternative. Here are some samples:

```
when before 4/12/92 # files before a date
when before 300 # files older than an age
when after 3/25 # or younger than a date this year
when before 2/1/94 and after 10/31/93 # even a range
```

More options and clauses are supported. Look at the code for clues. This one only works in the current directory.

Requires: UNIX. (CT)

---

xtable

`xtable` prints various character translation tables. See procedure `help()` for the capabilities.

Links: options, colmize, hexcvt, ebcdic. (RJA, AB)

---

yahtz

This classic dice game will run under UNIX and under DOS as well. It should run out of the box on DOS as long as you stay in the current directory. See the README file.

Links: `iolib`, `random`. (CT, RLG, PLT)

---

zipsort

`zipsort` sorts labels produced by labels in ascending order of their postal zip codes. Option: The option `-d n` sets the number of lines per label to n. The default is 9. This value must agree with the value used to format the labels. Zip Codes: 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 for extended zip codes. If a label does not end with a legal zip code, it is placed after all labels with legal zip codes. In such a case, an error messages also is written to standard error output.

Links: `options`. See also: `labels.icn`. (REG)

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?pyaho
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: Portability Considerations

Unicon's POSIX−based system interface is divided into two pieces. The facilities presented in Chapter 5 are portable. You can expect the portable system interface to be available on any implementation of Unicon. Facilities will be implemented as completely as is feasible on each platform. This appendix presents the rest of the Unicon POSIX interface, as well as some notes on functionality specific to Microsoft Windows.

POSIX extensions

Technically, the extensions presented in this section may or may not be part of the POSIX standard. The important issue is that 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 system files, `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. When 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 an additional, lower−level interface using the functions `fork()` and `exec()`. `fork()` causes the system to make a copy of the current process. After the fork there will be two identical processes that share all resources like open files, and differ only in one respect: the return value they each got 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.

Quite often `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].

```
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(f, u:−1, g:−1) : null?

`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) : null?

`chroot(f)` changes the root of the filesystem to `f`.

---

crypt(string, string) : string

`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

`exec(s, arg0, arg1, arg2, ...)` *replaces* the currently executing Icon program with a new program named in argument `s`. The remaining 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)

`fcntl(file, cmd, arg)` performs a number of 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)

l 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 will be returned by F_GETLK:

```
record posix_lock(value, pid)
```

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) : null?

`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

`filepair()` creates a bi−directional pair of connected files. This is analogous to the POSIX `socketpair(2)` function. It returns a list of two files, such that writes on one will be available on the other. The connection is bi−directional, unlike that returned by function `pipe()`. The two files are indistinguishable. 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

This function creates a new process that is identical to the current process in all respects 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

`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

`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

`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.

getgr(g) : record

`getgr(g)` returns a record that contains group file information for group `g`. `g` may be 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(string) : record

`gethost(h)` returns a record that contains host information for the machine named `h`. If `h` 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.

Return type: record posix_hostent(name, aliases, addresses)

getpgrp() : integer

`getpgrp()` returns the process group of the current process.

getpid() : integer

`getpid()` produces the process identification (pid) of the current process.

getppid() : integer?

`getppid()` produces the pid of the parent process.

getpw(u) : posix_password

`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

`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.

Return type: record posix_servent(name, aliases, port, proto)

If `proto` is defaulted, it will return the first matching entry.

getuid() : string

`getuid()` produces the real user identity (uid) of the current process.

kill(integer, x) : null?

`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) : null?

`link(src,dest)` creates a (hard) link `dest` that points to `src`.

setgid(integer) : null?

`setgid(g)` sets the group id of the current process to `g`. Consult the UNIX System V `setgid(2)` manual 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() : null?

`setpgrp()` sets the process group. This is the equivalent of `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) : null?

`setuid(u)` sets the user id of the current process to `u`. Consult the UNIX System V `setuid(2)` manual page.

symlink(string, string) : null?

`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`. umask integers encode nine bits for 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 return value is a string that represents the pid and the exit status as defined in this table:

UNIX equivalent	example of returned string
WIFSTOPPED(status)	"1234 stopped:SIGTSTP"
WIFSIGNALLED(status)	"1234 terminated:SIGHUP"
WIFEXITED(status)	"1234 exit:1"
WIFCORE(status)	"1234 terminated:SIGSEGV:core"

Currently the rusage facility is unimplemented.

Messaging Facilities

A suite of messaging facilities is available under several flavors of UNIX. The messaging facilities are based on the libtp messaging library by Steve Lumos. Messaging defines no new functions, but extends open() to handle several new types of connections using mode "m". The filename argument to a messaging connection is a URI (Uniform Resource Indicator, formerly denoted by URL) that indicates the protocol, 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, a string like "Reply To: jeffery@cs.unlv.edu" might be supplied as a third parameter to open() on an SMTP connection.

HTTP is used to read or write to Web servers; it typically consists of HTML text. Finger is used to query a finger server for information about users; it is not supported on many systems for security reasons. SMTP is used to send a mail message. POP is used to read mail messages. POP is the least file−like of these protocols, since a POP connection feels like a list of strings, each string containing an entire mail message, rather than a simple sequence of bytes..

Microsoft Windows

Windows versions of Icon may support non−portable system interfaces. Consult the `readme.txt` file and the on−line reference (Icon Project Document 271) for details.

Partial Support for POSIX

Windows supports the `getpid()` function, but not many of the other process−related functions such as `getppid()`.

Windows supports the `exec()` function in addition to Unicon's extended flavor of `system()`, but these functions may only launch Windows 32−bit .EXE binaries.

Windows Icon supports the following signals in functions such as `kill()`: `SIGABRT`, `SIGBREAK`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, and `SIGTERM`.

MS Windows Icon partially supports the `umask()` function, but ignores execute permission and treats user/group/world identically, using the most permissive access specified in the umask.

Native User Interface Components

Windows Icon 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)` allows the user to choose a file to open.

`WinPlayMedia(w, x[])` plays a multimedia resource.

`WinSaveDialog(w, s1, s2, i, s3, j)` 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.

Appendix D: Differences between Icon and Unicon

This appendix summarizes the known differences between Arizona Icon and Unicon.

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.

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.

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.

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.

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: About the CD−ROM

The CD−ROM that comes with this book contains a snapshot of every document and every implementation of Icon that is freely available on the Internet. These include:

* Icon 9.3.2 from the University of Arizona
* Jcon 2.0, also from the University of Arizona
* ProIcon for the Macintosh
* Unicon from the University of Nevada, Las Vegas

The first three implementations cover platforms including Acorn, Amiga, Macintosh, MS−DOS, MS Windows, OS/2, many flavors of UNIX, and VMS. The files are provided unmodified, as obtained from the University of Arizona Internet site in June of 1999. Browse the CD with your favorite web browser, and read the README files for installation instructions for these implementations. You may also wish to consult

```
http://www.cs.arizona.edu/icon/
```

to look for updates and additions to this software.

Icon and ProIcon are in the public domain. Jcon is copyrighted by the University of Arizona but freely redistributable. See the disc to view the copyright notice.

Unicon Online Resources

The implementation of Icon described in this book is Unicon. Unicon is copyrighted but freely distributable so long as its authors (Clinton Jeffery and Shamim Mohamed) are acknowledged. It is available on the Internet at

```
http://icon.cs.unlv.edu/
```

You may wish to consult this site for book errata, software updates, and support.

The current version of Ivib and the GUI toolkit can be found at the following location:

```
http://www.cpon-line.co.uk/rpp/gui.html
```

The current version of Iyacc and Iflex can be found at the following location:

```
http://icon.cs.unlv.edu/iyacc/
http://icon.cs.unlv.edu/iflex/
```

Unicon Installation from CD

Windows users will find the installation is launched from an AUTORUN.INF file when the CD is inserted. If the AUTORUN.INF does not execute for some reason, run the program *D*:\WINDOWS\SETUP.EXE, where *D* is your CD−ROM's drive letter.

Redhat Linux users will find RPM files in /*D*/redhat/, where *D* is the mount point for your CD−ROM.

Other flavors of UNIX rely on manual installation from gzip−compressed tar files. Consult the README files in the /*D*/bin and /*D*/src directories for instructions. In the worst case, if your platform is not supported you will have to build from source code. Changes to the source code may be required in order to port it to a new system; you may have to modify files to be compatible with your C compiler. Send e−mail to help@icon.cs.unlv.edu if you require assistance.

References

Barry W. Boehm. A Spiral Model of Software Development and Enhancement. IEEE Computer, vol. 21 no. 5, p. 61−72. 1988.

Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database Systems. The Benjamin/Cummings Publishing Company, Redwood City, CA. 1989.

Hans−Erik Eriksson and Magnus Penker. UML Toolkit. Wiley, New York, NY. 1998.

Ralph E. Griswold and Madge T. Griswold. The Icon Programming Language, 3rd ed. Peer−to−Peer Communications, San Jose, CA. 1996.

Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend. Graphics Programming in Icon. Peer−to−Peer Communications, San Jose, CA. 1998.

Clinton L. Jeffery. Program Monitoring and Visualization: An Exploratory Approach. Springer−Verlag, New York, NY. 1999.

John R. Levine, Tony Mason, and Doug Brown. lex & yacc. O'Reilly and Associates, Sebastopol, CA, 1990.

Larry Wall and Randall Schwartz. Programming Perl. O'Reilly and Associates, Sebastopol, CA, 1991.