From *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend, Peer-to-Peer Communications, Inc., 1998, ISBN 1-57398-009-9. Reprinted by permission of the publisher.

Copyright © 1998 Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend.

Chapter 2

Icon

This chapter addresses the basic concepts of Icon and describes many of its features. The chapter lays a foundation for chapters that follow, which describe Icon's graphics facilities. Some important aspects of Icon that do not fit well into a general description of its facilities are given in the **Special Topics** section at the end of this chapter. Be sure to read these.

You probably won't want to try to absorb everything in this chapter in one reading. Try reading through the chapter once without worrying about all the details, in order to get a "feel" for Icon. Later you may wish to read some sections more carefully or refer to them when reading the rest of this book or writing your own programs.

The appendices at the end of this book contain detailed reference material, including parts of Icon's computational repertoire that are not described elsewhere in this book. If you need to do something that we don't describe here, look in the appendices on operations and procedures or pick up a copy of *The Icon Programming Language* (Griswold and Griswold, 1996).

Getting Started

Icon is an imperative, procedural programming language, similar in many respects to Pascal or C. A traditional first program that writes a greeting looks like this in Icon:

```
procedure main()
  write("Hi there!")                    # write a greeting
end
```

The words procedure and end are reserved words that indicate the beginning and ending of a procedure. Every program must have a procedure named main,

which is where execution begins. The word write is the name of a built-in procedure. When it is called, as it is here, it writes out its argument. Parentheses indicate a call and enclose arguments. In this case there is one argument, which is a string of characters indicated by the delimiting quotation marks. The character # begins a comment that continues to the end of the line.

Most Icon programs contain many procedures. The procedures declared in a program are on a par with built-in procedures and are called the same way, as in

```
procedure main()

  greet("George")

end

procedure greet(name)

  write("Hi there!")
  write("My name is ", name, ".")

end
```

which writes

```
Hi there!
My name is George.
```

The word name is a parameter of the procedure greet(); its value is set when greet() is called. In the example above, "George" becomes the value of name in greet(). Note that the second use of write() has three arguments separated by commas. When write() is given several arguments, it writes them one after another on the same line.

You may wonder why there are no semicolons following lines of code. Icon allows them but doesn't require them — it supplies them automatically so that you don't have to worry about them. The procedure greet() could be written as

```
procedure greet(name);
  write("Hi there!");
  write("My name is ", name, ".");
end;
```

but the semicolons are unnecessary.

We've been careful in our choice of words about semicolons. In Pascal, semicolons separate *statements* that do things, while *expressions* perform computations.

In Pascal, the following line is a statement:

```
if switch = on then write('on') else write('off');
```

In this statement, switch = on is an expression whose value determines what is written.

Icon is different in this regard. Icon has no statements, only expressions. Icon expressions look like statements in Pascal and do similar things. Every Icon expression, however, returns a value, and the value often is useful. In Icon, that statement could be cast as either of these expressions:

```
if switch = on then write("on") else write("off")
write(if switch = on then "on" else "off")
```

Although the second form is not better than the first from a stylistic standpoint, it shows that even if-then-else is an expression.

For the most part, when writing Icon programs, you'll just use expressions in natural ways without worrying about the difference between statements and expressions.

Expression Evaluation

At first glance, expression evaluation in Icon may appear to be the same as in other imperative programming languages. Although expression evaluation in Icon has some aspects of expression evaluation in other imperative languages, there's much more to Icon, as we'll show soon.

Sequential Evaluation

In Icon, as in most other imperative languages, expressions are evaluated in the order in which they are given, as in

```
name1 := read()
name2 := read()
write("The first two names are ", name1, " and ", name2, ".")
```

which reads two lines of input and writes out an informative line containing the results.

The sequential order of evaluation can be changed by a control structure, as in

```
if name1 == name2 then {
  scount := scount + 1
  write("The names are the same.")
  }
```

```
else {
  dcount := dcount + 1
  write("The names are different.")
  }
```

The expression name1 == name2 performs string comparison on the values of name1 and name2. Which count is incremented and what is written depends on whether or not the values are the same. The braces enclose compound expressions. In this example, there are two expressions to evaluate in each “arm” of the control structure.

Success and Failure

If you are familiar with Pascal, you might think that the comparison expression name1 == name2 produces true or false, which is then used by if-then-else to determine what to do.

In Icon, such a comparison expression does not produce a logical value; instead, it either *succeeds* or *fails*. The effect is the same in the example above, but the difference between logical values and success or failure is fundamental and important.

The idea behind success and failure is that sometimes a perfectly reasonably computational expression may not be able to produce a result. As an analogy, imagine turning a doorknob to open a door. If it opens, your attempt succeeds; it the door is locked, your attempt fails.

An example in programming is attempting to read a line from a file. In Icon, such an attempt succeeds if there is a line remaining in the file but fails if there are no more lines. For example,

```
while line := read() do
  write(line)
```

reads and writes lines until the end of the input file is reached. At that point, read() fails. When read() fails, there is no value to assign to line, no assignment is performed, the assignment fails, and the while-do loop is terminated. Note that the failure of read() is “inherited”by assignment — an assignment can’t be performed if there’s nothing to assign.

Since failure is inherited, this loop can be written more compactly as

```
while write(read())
```

The do clause and the auxiliary identifier are not needed.

One of the advantages of using success and failure instead of logical values to control the order of program execution is that any expression, not just

a logical one, can be used in a control structure. In addition, the notion of attempting a computation that may succeed or fail also is a natural analogy to the way we carry out our daily activities in getting around in the world.

If you're used to using logical expressions in programming, the success-and-failure approach may appear strange at first. As you get accustomed to it, you'll find it both natural and powerful.

As you learn Icon, pay attention to the situations in which expressions may fail. We've given two examples so far: comparison and reading input. There are many others, which we'll mention as we go along.

The general criterion for expression failure is a computation that is meaningful but can't be carried out in a particular instance. Some computations, however, are simply erroneous. An example is

```
i := i + "a"
```

which is an error and terminates program execution because "a" is not a number.

What Icon considers an error as opposed to failure is a matter of language design that tries to strike a balance between convenience and error detection. Once you get used to Icon, you won't have to worry about this. Instead, you'll find that failure is a convenient way of making decisions and controlling loops.

Success and failure of expressions in combination can be tested using *conjunction* and *alternation*. Both have a familiar appearance. Conjunction is expressed as

expr1 & *expr2*

which succeeds only if both *expr1* and *expr2* succeed. For example,

```
if (max > 0) & (min < 0) then write("The bounds are bracketed.")
```

writes a line only if max is greater than zero and min is less than zero.

Alternation is expressed as

expr1 | *expr2*

which succeeds if either *expr1* or *expr2* succeeds. For example,

```
if (pct < 0) | (pct > 100) then write("Invalid percentage.")
```

writes a line only if pct is less than 0 or greater than 100.

Control Structures

Icon has several control structures that can be used to determine which expressions are evaluated and in what order. Most control structures, including

if-then-else and while-do in the preceding section, use success or failure to determine what to do.

There are two looping control structures in addition to while-do:

until *expr1* do *expr2*
repeat *expr*

The control structure until-do is the opposite of while-do; it loops until *expr1* succeeds. The control structure repeat evaluates *expr* repeatedly; it does not matter whether *expr* succeeds or fails.

You can terminate any loop by using break, which exits the loop and allows evaluation to continue at the point immediately after the loop. For example,

```
while line := read() do
  if line == "stop" then break
  else write(line)
```

writes lines until one that is "stop" is encountered, at which point the loop is terminated.

It's also possible to go directly to the next iteration of a loop without evaluating the remaining portion of the do clause. This is done with next. For example,

```
while line := read() do
  if line == "skip" then next
  else write(line)
```

which doesn't write lines that are "skip". There's a better way to do this without using next:

```
while line := read() do
  if line ~== "skip" then write(line)
```

The operator ~== is the opposite of ==; s1 ~== s2 succeeds if s1 differs from s2 but fails otherwise. Although next is not needed in the loop shown above, in more complicated situations next often provides the best method of getting directly to the next iteration of a loop.

The control structure

not *expr*

succeeds if *expr* fails but fails if *expr* succeeds. In other words, not reverses success and failure. This control structure could have been called cant to emphasize the use of success and failure in expression evaluation in Icon.

Icon has one control structure in which the expression to evaluate is based on a value rather than success or failure:

```
case expr of {
  case clause
  case clause
      …
  }
```

The value of *expr* is used to select a case clause and an expression to evaluate. Case clauses are evaluated in the order they are given. A case clause has the form

```
expr1 : expr2
```

where the value of *expr1* is compared to the value of *expr* at the beginning of the case expression. If the value of *expr1* in a case clause is the same as the value of *expr*, *expr2* is evaluated and control goes to the point immediately after the end of the case expression. Otherwise, the next case clause is tried. For example, in

```
case title of {
  "president":  write("Hail to the chief!")
  "umpire":     write("Throw the bum out!")
  default:      write("Who is this guy?")
  }
```

if the value of title is "president" or "umpire", a corresponding line is written. If the value of title is neither of these strings, the default case is selected. The default case is optional; if it is absent and no case clause is selected, nothing happens.

Once a case clause is selected, no other case clauses are tried; unlike C's switch statement, control never passes from a selected case to the next one. Alternation can be used to let one of several values select the same case clause, as in:

```
case title of {
  "president" | "prime minister":       write("Hail to the chief!")
  "umpire" | "referee" | "linesman":    write("Throw the bum out!")
  default:                              write("Who is this guy?")
  }
```

Generators

Now for something fun: Imagine you are in a room with three closed doors and no other way out. Suppose you try a door and find it locked. You'd probably try another door. In other words, you are presented with three alternatives. If you try one and fail to get out of the room, you'd try another door, and, if necessary the third one.

Analogous situations are common in programming problems, but most programming languages don't provide much help. Icon does.

Consider the problem of locating the positions at which one string occurs as a substring in another. Suppose you're looking for the string "the" in a line of text. Consider three possible lines:

"He saw the burglar jump down."

"He saw a burglar jump down."

"He saw the burglar jump over the bench and climb the wall."

In the first line, there is one instance of "the", as shown by the underline. In the second, there is none, but in the third, there are three.

If you are looking for "the" in the first line, you would be successful. In the second line, you would fail — a situation we've already covered. But what about the third line, where there are three instances of "the"? Certainly your attempt to find "the" should be successful. Finding the first (left-most) one would be natural. But what about the two remaining alternatives? Icon provides help with this kind of situation with *generator*s, which are expressions that can produce more than one value.

Icon has a procedure for finding the location of substrings: find(s1, s2), which produces (generates) the positions at which s1 occurs in s2. Suppose we name the lines above line1, line2, and line3. Then find("the", line1) produces 8 (we'll explain how Icon numbers the positions in strings later). On the other hand, find("the", line2) fails, since there is no occurrence of "the" in line2. find("the", line3) produces 8, then 30, and finally 50.

A generator does not produce several values all at once. Instead, a generator produces a value only when one is needed. For example, in

```
i := find("the", line1)
```

find("the", line1) produces 8 because a value is needed for the assignment. As a result, 8 is assigned to i. On the other hand, in

```
i := find("the", line2)
```

since find("the", line2) fails, the assignment is not done, and i is not changed. Incidentally, it's a good idea to provide a test when there is a possibility of failure; otherwise you have no way of knowing if a computation was done.

Now let's consider the third line. In

```
i := find("the", line3)
```

the first position, 8, is assigned to i; find() works from left to right as you'd expect. Since assignment needs only one value to assign, only one value is produced by

find(). But what about the other two positions? Suppose you want to know what they are?

Generators wouldn't be much good if there weren't ways to get more than a first value. There are two ways, however: *iteration* and *goal-directed evaluation*.

Iteration

The control structure

```
every expr1 do expr2
```

requests every value that *expr1* can produce, evaluating *expr2* for each one. For example,

```
every i := find("the", line3) do
  write(i)
```

writes 8, 30, and 50. The loop is terminated when find() has no more values to produce.

Generation, like failure, is "inherited". The loop above can be written more compactly as

```
every write(find("the", line3))
```

You might try to write the equivalent computation in Pascal or C — that will show you the power of generators.

Although every requests all the values of a generator, you can put a limit on the number of values a generator can produce. The *limitation* control structure,

```
expr \ i
```

limits *expr* to at most i results. For example,

```
every write(find("the", line3)) \ 2
```

writes only 8 and 30.

A word of warning: It's easy to confuse while-do with every-do because they appear to be so similar. The difference is that

```
while expr1 do expr2
```

repeatedly evaluates *expr1*, requesting only its first value each time through the loop, while

```
every expr1 do expr2
```

requests all the values *expr1* has. For example, if you write

```
while write(find("the", line3))
```

the value 8 is written over and over, in an endless loop. You'll probably not make this mistake often, but it may be helpful to know what to look for if you get such a symptom.

The other mistake is to use every-do when you want to repeatedly evaluate an expression, as in

```
every write(read())
```

which writes (at most) one line, since read() is not a generator and can produce only one value. (If you're wondering why read() is not a generator, there's no need for it to be, since every time it is evaluated, it reads a line.)

Goal-Directed Evaluation

As mentioned earlier, there is a second way in which a generator can produce more than one value. It's called goal-directed evaluation, and unlike iteration, it's done automatically.

Suppose you choose a door in the imaginary room, but find that it opens to a closet with no exit. What you'd normally do is back out and try another door. You can imagine other, more complicated, situations in which you open a door into another room, it also has several doors, and so on, but you eventually wind up in a closet again.

The usual way to solve such problems is to be goal-directed; if something doesn't work, try something else until you succeed or exhaust all alternatives. If you are successful in solving a sub-goal (such as finding an unlocked door in the room you're currently in), but that doesn't lead to your ultimate goal (such as getting out of the place altogether), you go back and try another alternative (called backtracking). Of course, you have to keep track of what you've tried and not wind up repeating the same futile attempts. This can quickly become a problem, as in a maze.

In Icon, if a value produced by a generator does not lead to success in the expression that needed the value, the generator is automatically requested to produce another value (that is, to provide an alternative).

For example, suppose you want to know if "the" occurs in line3 at a position greater than 10. You can write

```
if find("the", line3) > 10 then write("Found it!")
```

As shown above, find() first produces 8. Since 8 is not greater than 10, the comparison fails. Things do not stop there, however. Goal-directed evaluation seeks success. The failure of the comparison results in a request for another value

from find(). In the case here, the next value is 30, the comparison succeeds, and a notification is written. All this happens automatically; it's part of expression evaluation in Icon.

You may have a lot of questions at this point, such as "What happens if there is more than one generator in a complicated expression?" and "Can't goal-directed evaluation result in a lot of unnecessary computation?"

We won't go into multiple generators here, except to say that all possible combinations of generators in an expression are tried if necessary. This sounds like an expensive process, but it's not a problem in practice. See Griswold and Griswold (1996) for a detailed discussion of multiple generators.

Reversible Assignment

When goal-directed evaluation results in backtracking, expression evaluation returns to previously evaluated expressions to see if they have alternatives.

Backtracking does not reverse the effects of assignment. For example, in

```
(i := 5) & (find("the", line) > 5)
```

if find() fails, backtracking to the assignment does not change the value assigned to i. It remains 5.

Icon provides reversible assignment, represented by <–. In

```
(i <– 5) & (find("the", line) > 5)
```

if find() fails, backtracking to the reversible assignment causes the value of i to be restored to whatever it was previously.

Other Generators

As you might imagine, Icon has several generators as well as a way for you to write your own. We'll mention generators that are specific to particular kinds of computation as we get to other parts of Icon. There are two generally useful generators that we'll describe here.

One is

```
i to j
```

which generates the integers from i to j in sequence. For example,

```
every i := 1 to 100 do
  lookup(i)
```

evaluates lookup(1), lookup(2), …, lookup(100). This can be written more compactly as

```
every lookup(1 to 100)
```

There is an optional by clause in case you want an increment value other than 1, as in

```
every lookup(0 to 100 by 25)
```

which evaluates lookup(0), lookup(25), lookup(50), lookup(75), and lookup(100).

Alternation, described earlier, is a generator:

expr1 | *expr2*

This expression first generates the values of *expr1* and then generates the values of *expr2*. For example,

```
every lookup(1) | lookup(33) | lookup(57)
```

evaluates lookup(1), lookup(33), and lookup(57). This can be written more compactly by putting the alternatives in the argument of lookup():

```
every lookup(1 | 33 | 57)
```

In this example, the arguments of alternation are just integers and produce only one value each. As suggested above, the expressions in alternation can themselves be generators. Going back to an earlier example,

```
every write(find("the", line1) | find("the", line2) | find("the", line3))
```

writes 8 (from line1), nothing from line2, and then 8, 30, and 50 from line3. This expression can be written more compactly by putting the alternation in the second argument of find():

```
every write(find("the", line1 | line2 | line3))
```

Types, Values, and Variables

Data types

Icon supports several kinds of data. Integers and real (floating-point) numbers are familiar. In Icon, strings — sequences of characters— also are a type of data. Strings are a fundamental data type that can be arbitrarily long. Strings in Icon are not arrays of characters as they are in most programming languages. Icon also has a data type for sets of characters in which the concept of membership is important. In Icon, several kinds of structures also are data values. We'll say more about the different types of data as we go along.

Variables

Most programming languages, including Icon, have variables to which values can be assigned. Icon, unlike most programming languages, does not limit a variable to one type of data. In Icon, variables are not typed but values are. That may sound a bit strange, but what we mean is illustrated by the procedure type(), which returns the name of the type of its argument. For example,

```
type(a + b)
```

returns either "integer" or "real", depending on the types of a and b. You might want to make a mental note about type() — it's handy for several purposes, including debugging.

Since variables are not typed, a value of any type can be assigned to any variable. For example, it's possible to assign an integer to a variable at one time and a string to the same variable at another time, as in

```
x := 1
  ...
x := "Hello world"
```

Although Icon lets you do this, it's generally better style to use variables in a type-consistent way. There are situations, which we will describe later, when the flexibility that Icon offers in this regard is very useful.

Keywords

Icon *keywords*, identified by names beginning with an ampersand, play a variety of special roles. Some, such as &pi and &e, provide constant values — in this case the mathematical constants π and e. Others, such as &date and &version, supply environmental information. A few keywords can be assigned values; an example is &random, the seed for random numbers. Keywords are listed in Appendix F.

Assignment

As shown in earlier examples, := is Icon's assignment operator. *Augmented assignment* combines assignment with another operation. For example,

```
i := i + 3
```

can be written as

```
i +:= 3
```

Most binary operations can be combined with assignment in this manner.

The exchange operator, :=:, interchanges the values of two variables. After execution of

```
x :=: y
```

x contains the previous value of y and y contains the previous value of x.

Type Checking and Conversion

Since variables are not typed, there are no type declarations in Icon. This has advantages; it saves writing when you're putting a program together. On the other hand, without type declarations, errors in type usage may go unnoticed.

Although Icon does not have type declarations, it's a strongly typed language. During program execution, every value is checked to be sure that it is appropriate for the context in which it is used. For example, as mentioned earlier, an expression like

```
i := i + "a"
```

results in an error when executed because "a" cannot be converted to a number.

Icon does more than just check types during program execution. When necessary, Icon automatically converts a value that is not of the expected type to the type that is expected. Real, integer, string, and character set values are converted in this manner. For example, in

```
i := i + "1"
```

the string "1" is automatically converted to the integer 1, since addition requires numbers.

While you're not likely to write such expressions explicitly, there are many situations in which automatic type conversion is convenient and saves you the trouble of having to write an explicit conversion. We've used that earlier in this chapter without comment. Suppose you want to count something and then write out the results. You can do it like this:

```
count := 0
  ...                # count items
write(count)
```

The procedure write() expects a string, so the integer value of count is automatically converted to a string.

It's also possible to convert one type to another explicitly, as in

```
i := integer(x)
```

The procedure integer() converts its argument to an integer if possible. If the conversion can't be performed, integer() fails, as you should expect from our earlier discussion of the situations in which failure can occur.

There are similar procedures for other data types. See Appendix E.

The Null Value

The null value is a special value that serves several purposes. It has a type of its own and cannot be converted to any other type. The keyword &null has the null value.

The null value can be assigned to a variable, but it is illegal in most computations. Variables are initialized to the null value, so the use of a variable before another value has been assigned to it generally results in an error.

The operations /x and \x can be used to test for the null value. /x succeeds and produces x if x has the null value. \x succeeds and produces x if x has a nonnull value. Since these operations produce variables, assignment can be made to them. For example,

```
/x := 0
```

assigns 0 to x if x has the null value, and

```
\x :=  0
```

assigns 0 to x if x has a nonnull value.

Numerical Computation

Graphics programming, even for simple drawings, involves a lot of numerical computation. Icon has the usual facilities for this.

Integer and Real Arithmetic

Integers in Icon are what you'd expect, except possibly for the fact that there is no limit on the magnitude of integers. You probably won't have much occasion to use integers that are a thousand digits long, but it may be helpful to know that you don't have to worry about integer overflow.

Real numbers are represented by floating-point values, and hence their magnitudes and precision depend somewhat on the platform you're using.

Integers can be represented literally in the ways we've shown earlier. Real numbers can be represented literally in either decimal or exponential form, as in 0.5 and 5E–1.

The standard mathematical operations are provided for both integer and real arithmetic:

`–n`	negative of `n`
`n1 + n2`	sum of `n1` and `n2`
`n1 – n2`	difference of `n1` and `n2`
`n1 * n2`	product of `n1` and `n2`
`n1 / n2`	quotient of `n1` and `n2`
`n1 % n2`	remainder of `n1` divided by `n2`
`n1 ^ n2`	`n1` raised to the power `n2`

In "mixed-mode" arithmetic, in which one operand is an integer and the other is a real number, the integer is converted to a real number automatically and the result is a real number.

It's worth noting that the sign of `n1 % n2` is the sign of `n1`.

Arithmetic operations group in the usual way, so that `a * b + c / d` is interpreted as `(a * b) + (c / d)`. Grouping is discussed in more detail under **Special Topics** at the end of this chapter.

Division by zero is an error, as are expressions such as

```
–1 ^ 0.5
```

which would produce an imaginary result.

The standard numerical comparison operations are available also:

`n1 = n2`	`n1` equal to `n2`
`n1 > n2`	`n1` greater than `n2`
`n1 >= n2`	`n1` greater than or equal to `n2`
`n1 < n2`	`n1` less than `n2`
`n1 <= n2`	`n1` less than or equal to `n2`
`n1 ~= n2`	`n1` not equal to `n2`

A successful comparison operation returns the value of its right operand. Consequently, the expression

```
i < j < k
```

succeeds and produces the value of `k` if and only if `j` is strictly between `i` and `k`.

Mathematical Procedures

Many drawings, even simple ones, require mathematical computations. Icon provides several procedures for performing trigonometric and other common mathematical computations:

sqrt(r)	square root of r
exp(r)	*e* raised to the power r
log(r1, r2)	logarithm of r1 to the base r2
sin(r)	sine of r in radians
cos(r)	cosine of r in radians
tan(r)	tangent of r in radians
asin(r)	arc sine of r in the range $-\pi/2$ to $\pi/2$
acos(r)	arc cosine of r in the range 0 to π
atan(r1, r2)	arc tangent of r1 / r2 in the range $-\pi$ to π
dtor(r)	radian equivalent of r degrees
rtod(r)	degree equivalent of r radians

See Appendix E for details.

Random Numbers

Random numbers often are useful for providing a little variety or a touch of the unexpected in otherwise mundane operations.

The operation ?i produces a random number. If i is positive, the result is an integer in the range $1 \le j \le i$. If i is 0, the result is a real number in the range $0.0 \le r < 1.0$. Random numbers in this range provide a convenient basis for scaling to any range.

Icon also has ways of randomly selecting from a collection of values. We'll mention these in the sections that follow.

Structures

In Icon, a structure is a collection of values. Different kinds of structures provide different organizations and different ways of accessing values. Icon has four kinds of structures: records, lists, sets, and tables.

Records

Icon's records are similar in some respects to Pascal records and C structs. A record has a fixed number of fields whose values are accessed by name. A record type must be declared, as in

```
record point(x, y)
```

which declares point to be a record type with two fields, x and y. This declaration also creates a *record constructor*, which is a procedure that creates instances of the

record. For example,

```
P := point(0, 100)
```

creates a "point" whose x field is 0 and whose y field is 100 and then assigns the result to P. A record declaration also adds a type to Icon's built-in repertoire, so that you can tell what the type of a record is. For example,

```
write(type(P))
```

writes point.

A field of a record is accessed by following the record with a dot and the field name, as in

```
P.x := 300
```

which changes the x field of P to 300.

A record can contain any number of fields, and a program can contain any number of record declarations. Different record types can have the same field names, as in

```
record square(label, x, y, w, h)
```

Icon determines the correct field from the type at execution time. For example, obj.x references the first field if obj is a point but the second field if obj is a square.

Lists

In Icon, a list is a sequence of values — a one-dimensional array or a vector. Icon's list data type is very flexible and is particularly useful in graphics programming.

You can create a list by specifying the values (*elements*) that the list contains, as in

```
colors := ["cyan", "magenta", "yellow", "black"]
```

which creates a list with the four elements shown.

You also can create a list of a specified size and provide a single value for every element, as in

```
coordinates := list(1000, 0)
```

which creates a list of 1000 elements, all of which are zero. List size is limited only by the amount of available memory.

Both [] and list(0) create an empty list with no elements. We'll show why you might want an empty list later.

The operator *L produces the size of a list (the number of elements in it). For example, *colors produces 4.

The value of an element can be obtained by subscripting it by position, as in

```
write(colors[3])
```

which writes yellow, the third element of colors. Note that Icon numbers list elements starting at 1. The value of an element of a list can be set by assigning to the subscripting expression, as in

```
coordinates[137] := 500
```

which sets the 137th element of coordinates to 500. A subscripting expression fails if the subscript is out of range. For example, colors[5] fails.

The element-generation operator, !L, generates all the elements in L from first to last. For example,

```
every write(!colors)
```

writes cyan, magenta, yellow, and black. You can even use the element-generation operator to set the elements in a list, as in

```
every !coordinates := 100
```

which sets all of the elements in coordinates to 100.

Another operation that sometimes is convenient is ?L, which selects an element of the list L at random. For example,

```
write(?colors)
```

writes one of the elements of colors.

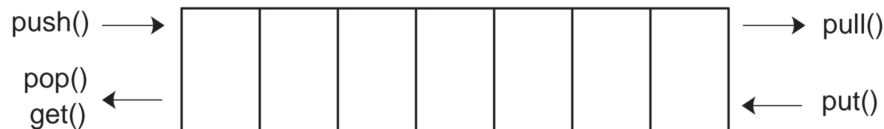
An unusual but very useful feature of lists in Icon is that you can use them as stacks and queues, adding and deleting elements from their ends. When these procedures are used, the size of a list increases and decreases automatically.

There are five procedures that access lists in these ways:

put(L, x1, x2, … xn)	puts x1, x2, … xn on the right end of L. The elements are appended in the order given, so xn becomes the last element of L.
push(L, x1, x2, … xn)	pushes x1, x2, … xn onto the left end of L. The elements are prepended in the order given, so that xn becomes the first element of L.
get(L)	removes the left-most element of L and produces its value. get() fails if L is empty.

pop(L)	pop() is a synonym for get().
pull(L)	removes the right-most element of L and produces its value. pull() fails if L is empty.

The relationships among these procedures are shown in the following diagram:



We mentioned empty lists earlier. If you want to implement a stack, you can start with an empty list and use push() and pop() on it. You can tell the stack is empty when pop() fails. To implement a queue, you also can start with an empty list but use put() and get(). You do not need to worry about overflow, since there is no limit to the size of a list.

These procedures also are useful even when you're not thinking of stacks and queues. For example, suppose you want to create a list of the lines from a file. All that's needed is

```
lines := []
while put(lines, read())
```

You don't need to know in advance how many lines are in the file.

Sets

A set in Icon is a collection of distinct values. In a set, unlike in a list, there is no concept of order and no possibility of duplicate values; only membership counts.

A set is created as follows:

```
shapes := set()
```

assigns to shapes an empty set (one with no members). Members can be added to a set, as in

```
insert(shapes, "triangle")
```

which adds the string "triangle" to shapes. The size of a set increases automatically as new members are added to it. Attempted insertion of a duplicate value succeeds without changing the set. There is no limit to the size of a set except the amount of available memory.

You can determine if a value is a member of a set as follows:

```
member(shapes, "square")
```

succeeds if "square" is in shapes but fails otherwise. You also can delete a member from a set, as in

```
delete(shapes, "triangle")
```

Attempting to delete a member that is not in a set succeeds but does not change the set.

The following set operations are available:

S1 ++ S2	produces a new set with the members that are in either S1 or S2 (union)
S1 ** S2	produces a new set with members that are in both S1 and S2 (intersection)
S1 -- S2	produces a new set with the members of S1 that are not in S2 (difference)

Many of the operations on lists also apply to sets: *S is the number of members in S, !S generates the members of S (in no predictable order), and ?S produces a randomly selected member of S.

Tables

Tables are much like sets, except that an element of a table consists of a key and an associated value. A key is like a member of a set — all the keys in a table are distinct. The values of different keys can be the same, however.

A table is created as follows:

```
attributes := table()
```

which assigns an empty table (one with no keys) to attributes. Elements can be added to a table by subscripting it with a key, as in

```
attributes["width"] := 500
```

which associates the value 500 with the key "width" in the table attributes. A new element is created if the key is not already present in the table. Note that this is much like assigning a value to an element of a list, except that the subscript here is a string, not a position. A table automatically grows as values are assigned to new keys. There is no limit to the size of a table except the amount of available memory.

As you'd expect, you can get the value corresponding to a key by subscripting. For example,

```
write(attributes["width"])
```

writes 500. You also can change the value associated with a key by assignment, as in

```
attributes["width"] := 1000
```

A default value is associated with every table. This value is fixed at the time the table is created and is specified by the argument to the table() call. If no argument is given, the null value is used for the table default.

When a table is subscripted by a value that does not match a key, the expression does not fail, but instead produces the table's default value. Continuing the example above,

```
attributes["height"]
```

succeeds and produces the null value because that is the table's default value. An expression such as /T[k] can be used to test whether k has been used to assign a value in T.

A default value of 0 is useful for tables that accumulate counts. For example, if

```
count := table(0)
```

then an expression such as

```
count["angle"] +:= 1
```

increments the value associated with "angle" whether or not it is the first time count is subscripted with this key.

The same operations that apply to lists and sets apply to tables: *T is the number of elements (keys) in T, !T generates the values (not keys) in T (in no predictable order), and ?T produces a randomly selected value from T. In addition, key(T) generates the keys in T (in no predictable order).

Sorting Structures

A structure can be sorted to produce a list with elements in sorted order. The details of sorting depend on the kind of the structure.

A list, set, or record can be sorted by sort(X), which produces a new list with the elements of X in sorted order. Sorting for numbers is in order of nondecreasing magnitude. Sorting for strings is in nondecreasing lexical (alphabetical) order. See Appendix E for details about sorting.

Sorting tables is more complicated because a table element consists of a pair of values. The way a table is sorted depends on the second argument of sort():

sort(T, 1) produces a list of two-element lists, where each two-element list corresponds to an element of T. Sorting of the two-element lists is by key.

sort(T, 2) is like sort(T, 1) except that the two-element lists are sorted by value.

sort(T, 3) produces a list of alternating keys and associated values. The resulting list has twice as many elementsas T. Sorting is by keys.

sort(T, 4) is like sort(T, 3), except that sorting is by value.

Characters and Strings

Characters are the material from which text is formed. Icon uses an 8-bit character set, which contains 256 characters. Characters are represented in a computer by small nonnegative integers in the range 0 to 255. These numbers are called the character codes. Although you ordinarily do not need to think of characters in terms of the character codes that represent them, it's useful to know that operations on characters, such as comparison, are based on the values of character codes.

All modern computer systems use a superset of the ASCII character set. As you'd expect, the code for B is greater than the code for A, and the code for 2 is greater than the code for 1. In ASCII, the codes for lowercase letters are greater than the codes for uppercase ones. Codes for characters other than letters and digits are somewhat arbitrary, and the meaning of codes greater than 127 is system-dependent.

Some characters do not have symbols associated with them, but designate special functions; tabs, backspaces, and linefeeds are examples. Some characters have no standard associations with symbols or functions but are used for a variety of purposes depending on the application that uses them. All 256 characters can be used in Icon programs. Unlike C, the null character (which has code 0), is not reserved for a special purpose.

Data Types Composed of Characters

Icon has two data types based on characters: strings and character sets (csets).

A string is a sequence of characters. Strings are used for many purposes, including printed and displayed text and text stored in files. Strings in Icon are atomic data values, not arrays of characters. A string is a value in the same sense

an integer is a value. Strings can be constructed as needed during program execution. Space for strings is provided automatically, and strings can be arbitrarily long, limited only by the amount of available memory.

A cset is just a collection of different characters. Unlike strings, there is no concept of order in a cset and a character can only occur once in a given cset. Csets are useful in string analysis in which certain characters, such as punctuation marks, are of importance, but no character is more important than another.

Strings are represented literally by enclosing a sequence of characters in double quotation marks, as in

```
greeting := "Hello world!"
```

which assigns a string of 12 characters to greeting. Escape sequences are used for characters that cannot be represented literally. For example, "\n" is a string consisting of a linefeed character, "\^C" is a control-C character, and "\"" is a string consisting of one double quotation mark. See Appendix A for a description of escape sequences.

Csets are represented in a similar fashion, but with enclosing single quotation marks, as in

```
operators := '+-*/^%'
```

which assigns a cset of 6 characters to operators.

Several keywords provide predefined csets. Two of the most useful are:

&digits	the 10 digits
&letters	all upper- and lowercase letters

See Appendix F for other cset-valued keywords.

Operations on Strings

Icon has a large repertoire of operations on strings. Some operations are used to create strings, while others are used to analyze strings. We'll discuss string analysis in the next section.

The most fundamental way to construct a string is concatenation, s1 || s2, which creates a new string by appending the characters of s2 to those of s1. An example of concatenation is

```
salutation := greeting || " (I'm new here, myself.)"
```

which forms a new string consisting of the characters in greeting followed by those given literally.

The empty string, which is given literally by "", is useful when you're

building up a string by concatenation. For example,

```
text := ""
while line := read() do
  text := text || line || " "
```

builds up a string of all the lines of input with a blank following each line. (This probably isn't something you'd actually want to do. Although Icon lets you build long strings, a list of strings usually is easier to process.)

The operation ∗s produces the size of s — the number of characters in it. For example, the value of ∗salutation as given above is 36. Incidentally, ∗s is fast and its speed is independent of the size of s.

Icon provides several procedures that construct strings. The procedure reverse(s) returns a copy of s with its characters in reverse order. The procedure repl(s, i) produces the concatenation of i copies of s. The procedures left(s, i), right(s, i), and center(s, i) position s in a field of a length i. The procedure trim(s) removes trailing spaces from s. These procedures are described in more detail in Appendix E.

Although strings in Icon are not arrays of characters, you can get the individual characters of a string by subscripting it. For example,

```
write(text[1])
```

writes the first character of text.

In Icon, unlike C and other programming languages that represent strings by arrays of characters, character numbering starts at 1, not 0. Character positions actually are between characters. For example, the character positions in "Medusa" are:

```
  M   e   d   u   s   a
↑   ↑   ↑   ↑   ↑   ↑   ↑
1   2   3   4   5   6   7
```

Position 1 is before the first character and position 7 is after the last character.

In subscripting a string, s[i] is the character following position i. The substring consisting of the characters between two positions can be obtained by subscripting with the positions separated by a colon. For example, the value of "Medusa"[2:5] is "edu".

Nonpositive numbers can be used to identify the characters of a string relative to its right end:

```
  M   e   d   u   s   a
↑   ↑   ↑   ↑   ↑   ↑   ↑
–6  –5  –4  –3  –2  –1  0
```

Thus, "Medusa"[–5:–2] is another way of specifying the substring "edu". In subscripting, a position can be given in either positive or nonpositive form and the positions do not have to be in order — it's the characters between two positions that count.

You can assign to a substring of a string to change those characters. Suppose, for example, the value of name is "George". Then

```
name[1:3] := "J"
```

changes name to "Jorge". Assignment to the substring creates a new string, of different length, which then is assigned to name. The expression above really is just shorthand for

```
name := "J" || name[3:0]
```

Unlike programming languages in which strings are arrays of characters, Icon doesn't really change the characters of a string; it always creates a new string in such situations.

Strings can be compared in a manner similar to the comparison of numbers, but the operators are different and comparison is by character code from the left — by lexical order. The string comparison operations are:

s1 == s2	s1 lexically equal to s2
s1 >> s2	s1 lexically greater than s2
s1 >>= s2	s1 lexically greater than or equal to s2
s1 << s2	s1 lexically less than s2
s1 <<= s2	s1 lexically less than or equal to s2
s1 ~== s2	s1 lexically not equal to s2

The operation s1 == s2 succeeds if and only if s1 and s2 have the same size and are the same, character by character. In determining if one string is greater than another, the codes for the characters in the two strings are compared from left to right. For example, "apple" is lexically greater than "Apple" because the character code for "a" is greater than the character code for "A". If two strings have the same initial characters, but one is longer than the other, the longer string is lexically greater than the shorter one: "apples" is lexically greater than "apple".

String Scanning

Icon has a high-level facility for analyzing strings, called string scanning. String scanning is based on two observations about the nature of most string analysis:

1. It is typical for many analysis operations to be performed on the same string. Imagine parsing an English-language sentence, for example.

The parsing is likely to require many operations on the sentence to identify its components.

2. Many analysis operations occur at a particular place in a string, and the place typically changes as analysis continues. Again, think of parsing a sentence. Parsing typically starts at the beginning of the sentence and progresses toward the end as components are identified. Of course, if an initial analysis proves to be incorrect later on, the analysis may go back to an earlier position and look for an alternative (backtracking).

To simplify string analysis, string scanning maintains a *subject* on which analysis operations can be performed without explicitly mentioning the string being analyzed. String scanning also automatically maintains a position that serves as a focus of attention in the subject as the analysis proceeds.

A string scanning expression has the form

```
s ? expr
```

where s is the subject string and *expr* is a scanning expression that analyzes (scans) it. When a string-scanning expression is evaluated, the subject is set to s and the position is set to 1, the beginning of the subject. The scanning expression *expr* often consists of several subexpressions.

There are two procedures that change the position in the subject:

tab(i)	set position to i
move(i)	increment the position by i

Both of these procedures produce the substring of the subject between the position prior to their evaluation and the position after their evaluation. Both of these procedures fail and leave the position unchanged if the specified position is out of the range of the subject. This failure can be used for loop control, as in

```
text ? {
  while write(move(1)) do          # write a character
    move(1)                        # skip a character
  }
```

which writes the odd-numbered characters of text, one per line.

It is good practice to enclose the scanning expression in braces, as shown above, even if they are not necessary. This allows a scanning expression to be extended easily and prevents unanticipated problems as a result of grouping with other expressions.

You can't do much with just the procedures shown above. String analysis

procedures, which produce positions that are set by tab(), are necessary for most string scanning. The most useful analysis procedures are:

find(s)	return the position at which s occurs in the subject
upto(c)	return the position at which a character of c occurs in the subject
many(c)	return the position after a sequence of characters of c

These procedures all examine the subject starting at the current position and look to the right. For example, find("the") produces the position of the first occurrence of "the" either at the current position or to its right. As you'd expect, analysis procedures fail if what's being looked for doesn't exist.

Analysis procedures produce positions; they do not change the position — tab() is used for this. For example, the "words" in a string can be written out as follows:

```
text ? {
  while tab(upto(&letters)) do
    write(tab(many(&letters)))
  }
```

In this string scanning expression, upto(&letters) produces the position of the first letter in the subject and provides the argument for tab(), which moves the position to that letter. Next, tab(many(&letters)) moves the position to the end of the sequence of letters and produces that substring of the subject, which is written. (Our definition of a "word" is overly simple, but it illustrates the general method of string scanning.)

Another useful scanning operation is

```
=s
```

which sets the position in the subject to the end of s, provided s occurs at the current position. It fails otherwise. For example, to analyze only lines of input that begin with a colon, the following approach can be used:

```
while line := read() do {
  line ? {
    if =":" then
      ...                              # analyze rest of the line
    }
  }
```

There is more to string scanning than we have described here. If you need to do a lot of complex string analysis, see Griswold and Griswold (1996) for more information.

Procedures and Scope

Procedure Declarations

Procedures are the computational building blocks from which programs are composed. Procedures allow you to organize computation and divide your program into logical components.

As illustrated by the examples given earlier in this chapter, procedure declarations are bracketed by procedure and end. Within the declaration, there can be declarations for variables that are local to the procedure, expressions to be evaluated on the first call of the procedure, and expressions comprising the body of the procedure that are executed whenever the procedure is called:

```
procedure name(parameters)
  local declarations
  initial clause
  procedure body
end
```

The parameters provide variables to which values are assigned when the procedure is called, For example in

```
procedure max(i, j)

  if i > j then return i else return j

end
```

the parameters of max() are i and j. When the procedure is called, values are assigned to these parameters, as in

```
write(max(count, limit))
```

which assigns the value of count to i and the value of limit to j, as if the expressions

```
i := count
j := limit
```

had been evaluated.

The return expressions in this procedure return either the value of i or the value of j, depending on their relative magnitudes. The value returned becomes the value of the procedure call. In the example above, this value is written.

When a procedure call omits the value for a parameter, the null value is used. The procedure can check for a null value and assign an appropriate default.

Parameters are local to a procedure call. That is, when max() is called, the variables i and j are created for use in the call only. Their values do not affect any variables i and j that might exist outside the procedure call.

Additional local variables are declared using the reserved words local and static. Variables declared as local are initialized to the null value every time the procedure is called. Variables declared as static are initialized to the null value on the first call, but they retain values assigned to them from call to call.

Expressions in an initial clause are evaluated only once, when the procedure is called for the first time. An initial clause often is used to assign static variables a first-time value.

The following example illustrates the use of local and static variables and an initial clause:

```
procedure alpha_count(s)
  local count
  static alphnum

  initial alphnum := &letters ++ &digits

  count := 0

  s ? {
    while tab(upto(alphnum)) do {
      count := count + 1
      move(1)
      }
    }

  return count

end
```

In this procedure, the value for alphnum is computed the first time alpha_count() is called, but it is available to subsequent calls of the procedure.

Scope

The term *scope* refers to the portion of a program within which a variable is accessible. As explained earlier, parameters and declared local variables are accessible only within a call of the procedure in which they are declared.

Variables also can be declared to be global, in which case they are accessible to the entire program. For example

```
global status, cycle
```

declares status and cycle to be global and hence accessible to all procedures in the program.

Global declarations must be outside procedure declarations. It is good practice to put them before the first procedure declaration in a program so that they are easy to locate when reading the program.

In the absence of a global declaration for a variable, the variable is local to the procedure in which it appears. A local declaration is not required for the variable. Although local declarations are not required in such cases, it is good practice to use them. It makes their scope clear and prevents an undeclared variable from accidentally being global because of an overlooked global declaration.

Calling Procedures

Procedures are values, much like lists and sets are values. The names of procedures, both built-in and declared, are global variables. Unlike declared global variables, these variables do not have null values initially; instead they have procedure values. When you call a procedure, as in

```
max(count, limit)
```

it's the value of max that determines which procedure is called. Since max is a declared procedure, the value of max is that procedure, which is called.

When a procedure is called, the arguments in the call are passed by value. That is, the values of count and limit in the call above that are assigned to the variables i and j in max. The procedure max does not have access to the variables count and limit and cannot change their values.

In the examples shown so far, the values passed to a procedure are given explicitly in argument lists in the calls. Sometimes it's useful to pass values contained in an Icon list to a procedure. This is especially useful for procedures like write() that can take an arbitrary number of arguments. Suppose, for example, that you do not know when you're writing a program how many arguments there should be in a call of write(). This might occur if lines to be written consist of fixed-width fields, but you don't know in advance how many fields there will be.

In such cases, a procedure can be called with an (Icon) list of values instead of explicit arguments. This form of call is

```
p ! L
```

where p is a procedure and L is a list containing the arguments for p. For the situation above, this might have the form

```
fields := [ ]
every put(fields, new_field())
write ! fields
```

Since procedures are values, they can be assigned to variables. For example, if

```
format := [left, right, center]
```

then

```
format[i](data, j)
```

calls left(), right(), or center() depending on whether i is 1, 2, or 3.

Procedure Returns

As shown earlier in this chapter, a declared procedure can return a value using a return expression, such as

```
return i
```

A declared procedure also can fail (produce no value) just as a built-in operation can fail. This is done by using the expression fail instead of return. For example, in

```
procedure between(i, j, k)

  if i < j < k then return j
  else fail

end
```

the value of j is returned if it is strictly between i and k, but the procedure call fails otherwise.

A procedure call also fails if control flows off the end, as in

```
procedure greet(name)

  write("Hi there!")
  write("My name is ", name, ".")

end
```

Two lines are written and then the procedure call fails. It's good practice in such cases to include an explicit return to prevent failure from causing unexpected results at the place the procedure is called. The previous procedure might better be written

```
procedure greet(name)

  write("Hi there!")
  write("My name is ", name, ".")
  return

end
```

If return has no argument, the null value is returned.

A procedure also can generate a sequence of values in the manner of a built-in operation. This is done using the expression suspend, which returns a value but leaves the procedure call intact so that it can be resumed to produce another value. An example is

```
procedure segment(s, n)

  s ? {
    while seg := move(n) do
      suspend seg
    }

end
```

This procedure generates successive n-character substrings of s. For example,

```
every write(segment("stereopticon"), 3)
```

writes

```
ste
reo
pti
con
```

When the scanning expression terminates because move(n) fails, control flows off the end of the procedure and no more results are generated; that is, it fails when resumed for another value. A fail expression could be added at the end of this procedure, but it is conventional when writing generating procedures to omit the fail.

File Input and Output

Files

On most platforms, a file is just a string of characters stored on a disk or entered from the keyboard. A text file consists of lines that end with line terminators. When reading a line, the characters up to a line terminator are

returned as a string and the line terminator is discarded. When a line is written, a line terminator is appended. Line terminators vary from platform to platform, but since they are discarded and added automatically, you usually don't have to worry about them.

It's also possible to read and write characters in "binary" mode without regard to line terminators. Most graphics applications deal with text files, but if you need to deal with binary data, see the description of open() in Appendix E.

We've illustrated reading and writing lines of text in preceding examples without mentioning files. Three files always are available. They are the values of keywords:

&input	standard input
&output	standard output
&errout	standard error output

When read() is called without an argument, it reads lines from standard input. You also can use &input as the argument to read(), as in read(&input). Standard input usually comes from the keyboard but also can come from a disk file. The method of specifying a file for standard input depends on the platform.

When write() is called without specifying a file, lines are written to standard output. You also can specify &output as the first argument of write(), as in

```
write(&output, "Hello, world!")
```

Standard output usually goes to the screen of your monitor, but there are ways of having it stored for later use.

Standard error output by convention is where error messages, diagnostics, and so forth are written. To write to standard error output, use &errout as the first argument of write(), as in

```
write(&errout, "Your data is inconsistent.")
```

Like standard output, standard error output usually goes to the screen, but most platforms provide a way to separate standard output from standard error output.

You also can open other files for reading and writing. The procedure open(name, mode) opens the named file in the mode specified. The most commonly used modes are:

"r"	open the file for reading (the default)
"w"	open the file for writing

The procedure open() returns a value whose type is file. For example,

```
poem := open("thanotopsis.txt")
```

opens the file thanotopsis.txt for reading and assigns the corresponding file value to poem. This file value then can be used as the argument for read(), as in

```
while line := read(poem) do
  process(line)
```

Note that the word *file* is used in two different ways: as the name for a file that the operating system understands and as an Icon value.

The procedure open() fails if the file cannot be opened in the specified mode. This may happen for a variety of reasons. For example, if thanotopsis.txt does not exist or if it's protected against reading, the use of open() above fails. If this happens, no value is assigned to poem. If no other value has been assigned to poem, its value is null. A null value and an omitted argument are the same in Icon, so read(poem) is equivalent to read(). This is not an error; instead lines are read from standard input, which may have mysterious consequences. It therefore is very important when opening a file to provide an alternative in case open() fails, as in

```
poem := open("thanotoposis.txt") | stop("*** cannot open input file")
```

The procedure stop() writes its argument to standard error output and then terminates program execution. It is the standard way to handle errors that prevent further program execution.

Writing Lines

As illustrated by previous examples, if write() has several string arguments, they are written in succession on one line. A line terminator is appended after the last string to produce a complete line.

Sometimes it's useful to write the components of a line in the midst of performing other computations. For example, if you want to see the pattern of word lengths in a line, you might decide to replace every word by its length:

```
sizes := ""

line ? {
  while tab(upto(&letters)) do
    sizes ||:= *tab(many(&letters)) || " "
  }

write(sizes)
```

The result might be something like

```
4 1 5 7 11
```

You can avoid the concatenation by using the procedure writes(), which is like write(), except that it does not append a line terminator. The code fragment above could be recast using writes() as follows:

```
line ? {
  while tab(upto(&letters)) do
    writes(*tab(many(&letters)), " ")
  }
write()
```

The sizes and separating blanks are written successively, but without line terminators. The final write() with no argument provides the line terminator to complete the line.

Closing Files

The procedure close(name) closes the named file. Closing a file that is open for output assures that any data that may be in internal buffers is written to complete the file. It also prevents additional data being written to that file until it is opened again. Closing a file that is open for reading prevents further data from being read from that file.

When program execution terminates, whether normally by returning from the main procedure, because of stop(), or as the result of a run-time error, all files are closed automatically. It therefore is unnecessary to close files explicitly before terminating program execution.

Most platforms, however, limit the number of files that can be open simultaneously. If you exceed this limit, open() fails. If you're using many files in a program, it therefore is important to close a file when you're through with it.

Preprocessing

Icon provides a preprocessor that performs simple editing tasks as a source program is read. Values or code fragments can be substituted wherever a chosen name appears. Lines of code can be enabled or disabled conditionally, and additional source files can be imported. The preprocessor is so named because all this editing takes place before the source code is compiled.

Preprocessor directives are identified by a $ as the first character of a line, followed by a directive name. For example,

```
$define Margin 8
```

defines the value of Margin to be 8. Whenever Margin appears subsequently in

the program, 8 is substituted. For example, the line

```
x := Margin
```

is interpreted as if it had been written

```
x := 8
```

A definition can be removed, as in

```
$undef Margin
```

which removes the definition of Margin. A name can be redefined, but it must be undefined first, as in

```
$undef Margin
$define Margin 12
```

In all cases, a definition affects only the portion of the file following the place it appears.

There are a number of predefined names that depend on the platform on which you are running. For example, _MS_WINDOWS is defined if you're running on a Microsoft Windows platform.

The directive $ifdef *name* enables subsequent lines of code up to $endif only if *name* is defined. There may be a $else directive between the $ifdef and $endif directives to enable code if *name* is not defined. For example,

```
$ifdef _MS_WINDOWS
  pathsym := "\\"
$else
  pathsym := "/"
$endif
```

enables

```
pathsym := "\\"
```

if _MS_WINDOWS is defined but

```
pathsym := "/"
```

otherwise.

The $include directive copies a specified file into the program at the place where the $include appears. For example,

```
$include "const.icn"
```

inserts the contents of the file const.icn to replace the $include directive. File names that do not have the syntax of an Icon identifier must be enclosed in quotation marks, as shown above.

See Appendix B for more information about preprocessing.

Running Icon Programs

Compilation and Execution

Running an Icon program involves two steps: compiling the program to produce an executable file and then executing that file.

The way that these two steps are performed depends on the platform on which Icon is run. On some platforms, Icon runs from a visual interface using menus and so forth. On other platforms, Icon is run from the command line. User's manuals that describe how to run Icon are available for the different platforms. We'll use a command-line environment here to illustrate what's involved and the options that are available.

On the command line, compilation is performed by the program icont, which processes an Icon source file and produces an executable *icode* file, as in

```
icont app.icn
```

which compiles the program app.icn (files containing Icon programs must have the suffix .icn). Specifying the .icn suffix is optional; the following works just as well as the example above:

```
icont app
```

The name of the icode file produced by compiling an Icon program is based on the name of the Icon file. On UNIX platforms, the name is obtained by removing the suffix and is just app for the example above. For Microsoft Windows platforms, the .icn suffix is replaced by .bat, producing app.bat for the example above.

A program can be compiled and executed in one step by following the program name by –x, as in

```
icont app.icn –x
```

There are several command-line options that can be used to control icont. For example,

```
icont –o rotor app
```

causes the icode file to be named rotor (or rotor.cmd on Windows platforms). Such options must appear before the file name, unlike –x.

See Appendix O for more information about compiling and executing Icon programs.

Libraries

As illustrated earlier in this chapter, procedures can be declared to augment Icon's built-in repertoire. Such procedures can be placed in libraries so that they are available to different programs. Libraries play an important part in graphics programming, and many of the graphics procedures described in subsequent chapters are contained in libraries rather than being built into Icon.

A library is included in a program by using a link declaration. For example,

```
link graphics
```

links the procedures needed for most graphics applications.

Link declarations can be placed anywhere in a program except inside procedure declarations. It is good practice to place them at the beginning of a program where they are easy to find.

You can make your own libraries. To do this, you need to compile the files containing the procedures by telling icont to save its result in library format, called *ucode*. This is done with the command-line option –c, as in

```
icont –c drawlib
```

which produces a pair of ucode files named drawlib.u1 and drawlib.u2. (The .u1 file contains code for the procedures, while the .u2 file contains global information). This pair of files then can be linked by

```
link drawlib
```

in the program that needs procedures in drawlib.

Only the procedures that are needed by a program are linked into it; you can make libraries that contain many procedures without worrying about the space they might take in programs that don't need all of them.

Environment Variables

Icon's compilation environment can be customized using *environment variables*. These variables, which are set before icont is run, tell Icon where to look for things like libraries specified in link declarations.

The way that environment variables are set depends on the platform on which you are running. In a UNIX command-line environment, the way an environment variable typically is set is illustrated by

```
setenv IPATH "/usr/local/lib/ilib /usr/icon/ilib"
```

which sets the environment variable IPATH.

IPATH is used to locate library files given in link declarations. In this example, Icon looks in the directories

```
/usr/local/lib/ilib
```

and

```
/usr/icon/ilib
```

in that order. Icon always looks in the current directory first, so if your library ucode files are there, IPATH need not contain that directory.

The environment variable LPATH is similar to IPATH, but LPATH tells Icon where to look for files mentioned in $include preprocessor directives. (You may notice that the names IPATH and LPATH seem backward — IPATH for library files and LPATH for include files. The source of this potential confusion has historical origins and it's now too late to correct it.).

Other environment variables are read when an Icon program begins execution to configure memory and other aspects of execution. Consult the user's manual for your platform.

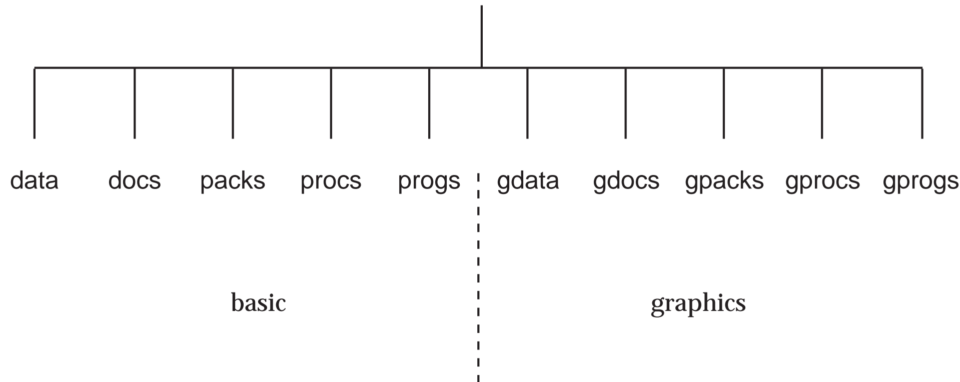
See Appendix O for more information about environment variables.

The Icon Program Library

The Icon program library is a free collection of programs, procedures, documentation, data, and support tools that is available to all Icon programmers. See Appendix P for instructions about obtaining the library.

Organization

The main directories in the Icon program library hierarchy are shown in Figure 2.1.



Icon Program Library Hierarchy **Figure 2.1**

The library has two main parts: basic material and graphics material. The initial character g indicates graphics material.

The source code for procedure modules is in the directories procs and gprocs. As one might expect, the source code for graphics is in gprocs. The directories progs and gprogs contain complete programs. The directories packs and gpacks contain large packages.

Core Modules

The directories procs and gprocs contain hundreds of files, and in these there are thousands of procedures. Some procedures are useful only for specialized applications. Others provide commonly used facilities and are designated as "core" procedures. The core modules for the basic part of the library are:

convert	type conversion and formatting procedures
datetime	date and time procedures
factors	procedures related to factoring and prime numbers
io	procedures related to input and output
lists	list manipulation procedures
math	procedures for mathematical computation
numbers	procedures for numerical computation and formatting
random	procedures related to random numbers
records	record manipulation procedures
scan	scanning procedures
sets	set manipulation procedures
sort	sorting procedures
strings	string manipulation procedures
tables	table manipulation procedures

Special Topics

This section contains information about aspects of Icon that may help you in writing and understanding Icon programs.

Syntactic Considerations

As in all programming languages, there are rules that you can follow to avoid syntactic problems. The worst problems are not those that produce syntax errors but those that produce unexpected results. The following sections deal with the most common sources of such problems in Icon programs.

Precedence and Associativity

Icon has many operators — more than most programming languages. The way that operators group in complex expressions in the absence of specific groupings provided by parentheses and braces depends on the precedences and associativities of the operators in such expressions.

Precedence determines which of two operators adjacent to an operand gets the operand. With one exception, prefix operators that come before their operands have precedence over infix operators that stand between their operands. For example,

```
–text + i
```

groups as

```
(–text) + i
```

The exception is record field references, in which the infix field operator has highest of all precedence. Consequently,

```
–box.line
```

groups as

```
–(box.line)
```

Different infix operators have different precedences. The precedences of infix arithmetic operators are conventional, with exponentiation (^) having the highest precedence; multiplication (∗), division (/), and remaindering (%) the next highest; and addition (+) and subtraction (–) the lowest. Consequently,

```
i ∗ j + k
```

groups as

```
(i ∗ j) + k
```

Icon has many infix operators, and it’s easy to get an unintended result

by relying on precedences for grouping. Instead, it's wise to use parentheses for the less-familiar operations, as in

```
heading || (count + 1)
```

The use of parentheses also makes it easier to read a program, even if you know what the precedences are.

Two common cases are worth remembering. Assignment has low precedence, so it's safe to write

```
i := j + k
```

knowing it groups as

```
i := (j + k)
```

In addition, conjunction has the lowest precedence of all operators, so it's safe to write

```
i > j & m > n
```

knowing it groups as

```
(i > j) & (m > n)
```

A word of warning: The string scanning operator has higher precedence than conjunction. Therefore

```
text ? tab(find(header)) & move(10)
```

groups as

```
(text ? tab(find(header))) & move(10)
```

which probably is not what's intended.

As a general rule, it's wise to enclose scanning expressions in braces to avoid such problems, as in

```
text ? {
  tab(find(header)) & move(10)
  }
```

This approach also makes it easy to add to scanning expressions and makes the scope of scanning clear.

Associativity determines which of two infix operators gets an operand between them. Most infix operators are left associative. For example,

```
i – j – k
```

groups as

```
(i – j) – k
```

(as is necessary for subtraction to work correctly).

The exceptions to left associativity are exponentiation and assignment. Thus,

```
i ^ j ^ k
```

groups as

```
i ^ (j ^ k)
```

as is conventional in mathematical notation.

Assignment also is right associative, so that

```
i := j := k
```

groups as

```
i := (j := k)
```

This allows a value to be assigned to several variables in a single compound assignment expression.

Line Breaks

As mentioned earlier, the Icon compiler automatically inserts semicolons between expressions on successive lines.

You can, however, continue an expression from one line to the next. To do this, you need to know how the compiler decides to insert semicolons. The rule is simple: If the current line ends a complete expression and the next line begins an expression, a semicolon is inserted. To continue an expression from one line to the next, just write it so that it's not complete on the current line. For example, in

```
i := j –
  k
```

the expression is continued to the second line, since the expression on the first line is not complete (an expression cannot end with an operator). On the other hand, in

```
i := j
  – k
```

a semicolon is inserted between the two lines, since the first line contains a complete expression and a minus sign is a valid way to begin a new expression.

A useful guideline when you want to continue an expression from one line to the next is to break the expression after an infix operator, comma, or left parenthesis.

Preprocessing

Icon's preprocessor allows a name to be assigned to an arbitrarily complicated expression. A simple example is

```
$define SIZE      width + offset
```

When SIZE is used subsequently in the program, width + offset is substituted for it.

Suppose SIZE is used as follows:

```
dimension := SIZE * 3
```

This groups as

```
dimension := width + (offset * 3)
```

where the obvious intention was

```
dimension := (width + offset) * 3
```

The value assigned to dimension almost certainly will be incorrect and result in a bug that may be hard to find — after all

```
dimension := SIZE * 3
```

looks correct.

The solution is easy: Use parentheses in the definition, as in

```
$define SIZE    (width + offset)
```

Then

```
dimension := SIZE * 3
```

is equivalent to

```
dimension := (width + 3) * 3
```

as intended.

Polymorphous Operations

Icon has a number of *polymorphous* operations; that is, operations that apply to more than one data type. For example, the prefix size operator, *, applies to many different data types: *X produces the size of X whether the X is a string, list, set, table, or record. Similarly, ?X produces a randomly selected element of X for these types, !X generates all the elements of X, and sort() works for several different types of data.

Polymorphism simplifies the expression of computations that are common to different types of data. It's worth keeping this in mind when writing

procedures; a procedure often can be written to work on different kinds of data. An example is this procedure for shuffling values:

```
procedure shuffle(X)

  every i := *X to 2 by –1 do
    X[?i] :=: X[i]

  return X

end
```

This procedure works for shuffling the characters of a string or the elements of a list or record.

Pointer Semantics

Icon's structures (records, lists, sets, and tables) have *pointer semantics.* A structure value is represented internally by a pointer — a "handle" that references the data in the structure. When a structure is assigned or passed through a parameter, the pointer is copied but not the data to which it points. This is as fast as assigning an integer.

Consider the procedure rotate(), which moves a value from the front of a list and places it at the end:

```
procedure rotate(lst)
  local v

  v := pop(lst)
  put(list, v)

  return

end
```

Then

```
nums : = [2, 3, 5, 7]
rotate(nums)
every write(!nums)
```

writes

```
3
5
7
2
```

Because the parameter lst points to the same data as the variable nums, rotate() modifies the contents of nums.

Sometimes the sharing of data is not wanted. For example, in

```
Tucson := ["Arizona", "Pima", 1883]
City := Tucson
```

both Tucson and City point to the same structure. Consequently, assigning to an element of City changes an element of Tucson, and vice versa. That may not be the intention.

The procedure copy(x) makes a copy of the structure x by duplicating the values to which it points. For example, after

```
City := copy(Tucson)
```

there are two different lists that can be modified independently.

The procedure copy() works this way only at the top level: Any structures in the data pointed to by x are not copied and remain shared by the original structure and its copy.

Another important ramification of pointer semantics structures is that (a pointer to) a structure can be an element of a structure. An example is

```
dates := [1492, 1776, 1812]
labels := ["discovery", "revolution", "war"]
lookup := [dates, labels]
```

in which lookup is a list that contains (points to) two other lists.

Pointers can be used to represent structures such as trees and graphs. For example, a node in a binary tree might be represented using a record declaration such as

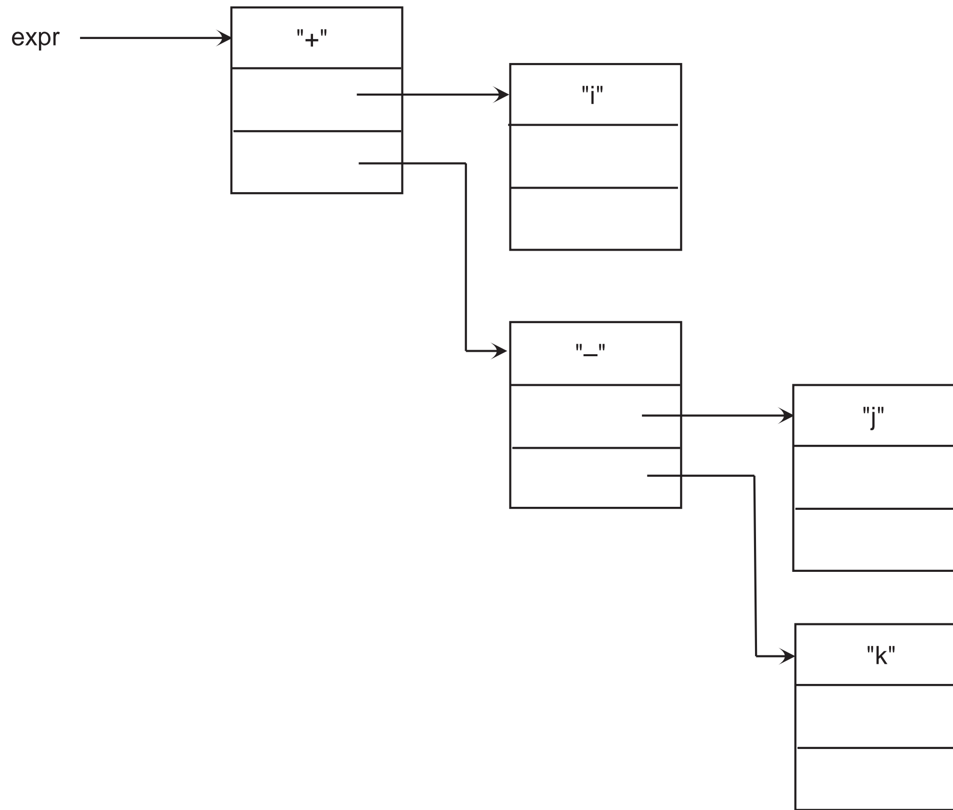
```
record node(symbol, ltree, rtree)
```

The field symbol contains a string for the contents of a node, while ltree and rtree are used as pointers to nodes for the left and right subtrees. For example,

```
expr := node("+", node("i"), node("–", node("j"), node("k")))
```

produces a binary tree. The omitted arguments default to null values and serve as “null pointers” in cases where there are no subtrees.

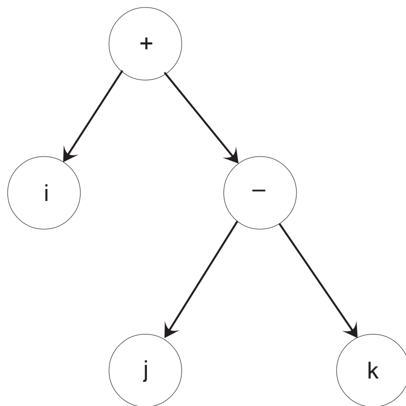
The structure that results can be visualized as shown in Figure 2.2.



A Record Structure **Figure 2.2**

The arrows emphasize the fact that structure values are pointers to blocks of data.

A more abstract representation is shown in Figure 2.3.



A Tree of Records

In this diagram, the details are omitted, leaving only what's needed to understand the structure.

Figure 2.3

Library Resources

The program library includes a whole directory full of nongraphical procedures. We can't even provide a concise summary, but here's a small sampling of what is available.

The strings module includes many procedures for manipulating strings, such as these:

replace(s1, s2, s3)	replace all occurrences of s2 in s1 by s3
rotate(s, i)	rotate s by i characters

The numbers module deals with things numerical:

gcd(i, j)	return greatest common divisor of i and j
roman(i)	convert i to roman numerals

Tips, Techniques, and Examples

Debugging

Debugging is one of the most difficult, time-consuming, and frustrating aspects of programming. Prevention is, of course, better than cure, but that's mostly a matter of good programming practice.

If you have a problem with a program, the easiest thing you can do is add write() expressions at judiciously chosen places to get more information. Although this is commonly done, it requires editing the program before and after finding the problem, and it also runs the risk of introducing its own errors.

If you do use write() expressions to get information about what is going on in a program, you may find it useful to use image(x) in the arguments of write(). The procedure image(x) produces a string representation showing the value and type of x. Using image() also is safe; write(image(x)) never produces an error, although write(x) will if x is not a string or a value convertible to a string.

Although adding write() expressions seems easy, you can get a lot of information about a program by tracing procedures. The keyword &trace can be used to give you information about procedure calls and returns. Setting &trace to –1 turns on procedure tracing and setting &trace to 0 turns it off. A word of warning: Trace output may be voluminous, especially in graphics programs that use library procedures.

Another way to get information is to set &dump to –1. This gives a listing of variables and values when program execution ends.

Even if you don't turn on procedure tracing or the termination dump, a run-time error produces a traceback of procedure calls leading to the expression in which the error occurred. It's often worth examining this traceback, rather than immediately looking in the program at the place the error occurred.

Often a more cerebral approach to debugging is faster and more effective than simply producing a lot of information in hopes of seeing something helpful. For Icon, there are a few common causes of errors that have recognizable symptoms that are worth checking before adding write() expressions or turning on tracing and the termination dump.

Incorrect data types are common causes of errors. In such cases, the error message on termination indicates the expected type and the actual value. The message procedure or integer expected accompanied by an "offending value" of &null usually occurs as a result of misspelling a procedure name, as in

```
wirte(message)
```

Since wirte presumably is a misspelling of write, wirte most likely is an undeclared identifier that has the null value when wirte(message) is evaluated. Hence the error.

You can go a long way toward avoiding this kind of error by doing two things: (1) declaring all local identifiers, and (2) using the –u option for icont, as in

```
icont –u app
```

This option produces warning messages for undeclared identifiers. In the example above, wirte probably will show up when icont is run, allowing you to fix the program before it is run.

Evaluating Icon Expressions Interactively

Although Icon itself does not provide a way to enter and evaluate individual expressions interactively, there is a program, qei, in the Icon program library that does. This program lets you enter an expression and see the result of its evaluation. Successive expressions accumulate, and results are assigned to variables so that previous results can be used in subsequent computations.

At the > prompt, an expression can be entered, followed by a semicolon and a return. (If a semicolon is not provided, subsequent lines are included until there is a semicolon.) The computation is then performed and the result is shown as an assignment to a variable, starting with r1_ and continuing with r2_, r3_, and so on. Here is an example of a simple interaction:

```
>   2 + 3.0;
    r1_ := 5.0
>   r1_ * 3;
    r2_ := 15.0
```

If an expression fails, qei responds with Failure, as in

```
>   1.0 = 0;
    Failure
```

The program has several other useful features, such as optionally showing the types of results. To get a brief summary of qei's features and how to use them, enter :help followed by a return.