Icon Programming Language Handbook

Thomas W. Christopher

Icon Programming Language Handbook

Thomas W. Christopher

Beta edition.

Published by Dr. Thomas W. Christopher, Tools of Computing LLC, P. O. Box 6335, Evanston IL 60204-6335.

Acknowledgment

I wish to thank Patricia Guilbeault for her technical editing of this document.

Todd Proebsting found several errors in the text and suggested improvements.

Printed in the United States of America.

CONTENTS

List of Figures

List of Tables

Preface

This document is designed to serve two purposes: to introduce to Icon and to be a reference for Icon.

As an introduction to programming in Icon, the handbook assumes you already know how to program in some other procedural programming language—C or Pascal, say. Some of the examples assume you know as much mathematics as the average college sophomore. Chapter 2 on page 21, presents features of the Icon programming Language equivalent to those of other programming languages.

To learn Icon, we suggest you read the Basics chapter and then read through the text and examples in the rest of the handbook.

Look things up in the tables as you need to, but don't get bogged down in them. The tables are there for reference. Just glance over them; don't bother to read them thoroughly when you are first learning the language.

As a reference on Icon, this may be the only document you have, so we include copious tables of operators, functions, and keywords. The tables are placed within discussions and examples of language features and language usage so that all the information you need will be close at hand. Since some language features fit into more than one topic, the tables have some overlapping material.

Near the end of the handbook we include tables of operators, functions, procedures, and keywords so you can quickly look up the specifics of functions, etc., when you remember their names.

On a personal note, the author confesses that he is not a "detail person," so it seems highly likely there will be some errors and omissions in this book. Please look me up on the world wide web and send corrections and suggestions.

-TC

www.iit.edu/~tc/
tc@charlie.cns.iit.edu

Chapter 1 About Icon

Icon is a very-high-level programming language. “Very-high-level” means roughly that it does a lot for you. It handles a lot of the details that you would have to handle for yourself if you were programming in a lower level language such as C; you can do more, quicker and more easily. This makes Icon ideal for:

Quick programming—If you need a program and you need it soon, Icon is a better choice than a lower level language.

Trying out ideas—If you have an idea for an algorithm, but you’re not sure it will work, it is better to try it out in Icon than invest a lot of time trying to get it going in some lower level language.

Prototyping— “There’s never enough time to do it right, but always enough time to do it over” is a cynical saying in the software field. Actually, that’s not such a bad approach. It’s only after you’ve implemented a system and used it for a while that you know what’s important and what isn’t, what works, and what doesn’t. It’s only then you understand it well enough to design it. Maybe the right way to do the system is to do it twice, and if that’s the case, why not do the first version in an easy, powerful programming language? The first version won’t cost as much if you do it in Icon.

Tools—Programmers often need small programs to do simple tasks. It’s a waste of time to expend much effort on these programs. You can get them out of the way much more quickly in Icon than in most other languages.

Text processing—Icon’s strings and tables make text processing much more convenient than in languages that only provide characters and character arrays.

Graphics programming—with version 9 of Icon came libraries of procedures for programming window and graphical interfaces. Icon makes graphical user interfaces easy.

General purpose programming—Well, why not?

The name, *Icon*, was chosen a long time before graphical user interfaces became popular. It does not refer to “icons,” but probably to *iconoclasm*, as the developers were excited about how their language diverged from current practices in language design.

There are some characteristics of Icon you will need to pay attention to as you learn it:

Very-high-level data types. Icon provides many built-in data types that you would have to program for yourself in other languages. The strings come with a powerful set of operations for text formatting. Even more useful are the tables, which provide simple, data-base-like facilities. You may need some practice before you discover their full power.Table 1 lists the built-in data types in Icon.

Table 1 Icon data types

Icon data type	Explanation
integer and real	The usual numeric data types. Some versions of Icon provide integers of unbounded precision.
null	There is only one value of this type, &null. Uninitialized variables are given this value.
string	Unlike most languages where strings are implemented as arrays of characters, Icon provides strings as a primitive data type. They can be of any length. There are extensive facilities for searching and editing strings.
cset (character set)	Character sets are used by string search functions to find or skip substrings of characters.
procedure	Procedures are values which can be assigned to variables.
list	Lists can be indexed like arrays. They can also be used as stacks and queues. All lists are dynamically allocated and can grow to any length the computer's memory can hold.
record	Records types can be declared. Each record type has a fixed number of named fields. They are used like records (or structs) in other languages. All records are dynamically allocated. In other languages they would be accessed by pointer, but since they are all accessed that way, there is no explicit pointer data type.
table	Tables associate values with keys. A value of any type can be used as a key. A value of any type can be used as a value.
set	A set is a collection of values of any type. Duplicate values are not represented, so no matter how many times a value is inserted into a set, it is present there only once.
co-expression	A co-expression is a part of the program running semi-independently from the other parts. It can be used as a generator to generate values from a sequence one at a time when needed, or it can be used as a co-routine, running as a concurrent process.
file	A file is an open file for reading or writing.
window	A window on the screen for interactive graphics.

Typelessness. Data values have data types. Variables do not. A value of any

type may be assigned to any variable. You will discover this to be quite useful when placing values of different types into the same lists, tables, and sets. However, you'll also discover that you will often make the mistake of using the wrong data type for an operation. Unlike many other languages, the Icon translator cannot tell you that you've made a mistake. Instead, you will get an error termination when you run the program.

Expression language. Icon is an expression language: almost all the executable constructs are expressions and can return values. There is no division between expression level constructs and statement-level constructs. You can nest control structures within expressions in a way that doesn't work in most other languages. You will find this very convenient at times. But you should be careful; it is easier to write highly complex expressions than to read them.

Goal-directed evaluation. The most difficult aspect of Icon for programmers familiar with other languages is its *goal-directed evaluation*, that is to say, its *backtracking control*. Most languages use a Boolean data type for controlling the flow of execution. In most languages, relational operators produce Boolean values which are tested by *if*'s and *while*'s. Icon is completely different. We'll spend a lot of time explaining how Icon works in Chapter 3. Briefly, in Icon expression evaluation can succeed or fail. If the expression succeeds, it produces a value. If it fails, control backs up to an expression it evaluated earlier to see if it will generate another value. If that expression does give another value, control starts forwards again to see if the later expression can succeed now.

Chapter 2 Basics

2.1 Compiling and running an Icon program

The author assumes you have version 9 of Icon installed on your system.

Suppose you want to call your Icon program “test”. You put your Icon program in a file “test.icn” and translate it with the command

```
icont test
```

If it translated without errors, you run it with the command

```
test
```

If there were errors, the Icon translator will tell you where it encountered the error and what the error seems to be. We will have a fuller discussion later, in Chapter 14.

2.2 Hello, world

It’s become customary to start off with a program that writes out “hello, world” to show that the translator is working. Here it is in Icon:

Figure 1 Hello world program

```
procedure main()
write("hello, world")
end
```

There are several things to notice:

You can see the way to write a procedure: it begins with `procedure` and ends with `end`. Following `procedure` is the name of the procedure and the parameter list, which can be empty.

Function `write` writes its arguments into the output and then terminates the line. The next write will begin on a new line.

Strings are enclosed in double quotes.

When the program begins running, it executes the procedure named `main`, just as in C.

In Figure 2 we show that in Icon either a new line or a semicolon—or both if you prefer—separate expressions in a sequence. Icon executes the expressions in a sequence in order. The function `writes` writes its arguments into the output but does not terminate the line. What is written next will follow on the same line.

Figure 2 Expression sequences.

```
procedure main()
writes(”hello,”)
write(” world”)
end

procedure main()
writes(”hello,”);
write(” world”)
end

procedure main()
writes(”hello,”); write(” world”)
end
```

Icon is a typeless language. That means that variables are not declared to have particular data types. Only values have data types, and a value of any type may be assigned to any variable. For that matter, variables do not have to be declared at all. Figure 3 illustrates this. Note the following:

Variable `x` is not declared at all.

The assignment operator is `:=`.

Variable `x` is assigned two values with different types, first a string and then an integer.

Procedure `write` is as willing to write out an integer as a string. In fact, it will write out anything it knows how to convert to a string.

Figure 3 Typelessness

```
procedure main()
x := "Example "
writes(x)
x := 1
write(x)
end
```

Figure 4 shows local declarations and the exchange operator. The two things to notice are:

The **`local`** introduces declarations of local variables within the procedure. They are allocated memory when the procedure is entered and they vanish when

the procedure returns. If there is no declaration for a variable, like `x` in Figure 3, the translator makes it a local variable.

Operator `:=:` is the exchange operator; it will exchange the values of two variables.

Figure 4 Local declarations

```
procedure main()
local x,y
x := " Example "
y := 2
write(x,y)
x :=: y
write(x,y)
end
```

When variables are created, they are given the initial value **`&null`** which causes most operations to report an error at run time. You will encounter that run-time error a lot.

Figure 5 Uninitialized variables

```
procedure main()
local x
write(x+1)#this will cause an error at run time
          # (and notice: comments begin with #
          # and run to the end of the line)
end
```

2.3 Declarations

We have seen local declarations. Icon actually provides all the following kinds of declarations:

Table 2 Declarations

Declaration	Example	Occurs	Explanation
local	`local x,y,z`	inside procedures	Creates new copies of the variables whenever the procedure is entered. Deletes them when the procedure returns. The names are known only within the procedure. Local is assumed for undeclared variables, but do not use this feature: introducing a global declaration later can cause a procedure to stop working.

Table 2 Declarations

Declaration	Example	Occurs	Explanation
static	`static x,y,z`	inside procedures	Creates copies of the variables when the program starts executing. The names are known only within the procedure. There is only one copy of a static variable. It retains its value between procedure calls.
global	`global x,y,z`	outside procedures	Creates copies of the variables when the program starts executing. The names are known only within all the procedures that do not declare the same names for local or static variables. There is only one copy of a global variable.

procedure	`procedure name(x,y,z)`
	`...`
	`end`
		outside procedures	See Section Chapter 5, Procedures, on page 53.

record constructor	`record name(x,y,z)`	outside procedures	See section 2.10 on page 33 and Chapter 12 on page 123.
linkage	`link name`	outside procedures	Tells the linker that this program uses procedures, records, or global variables declared in the file named *name*. The name may be an Icon identifier, but it must be a quoted string if it contains characters that Icon does not allow in identifiers.

2.4 Exiting a program

There are several ways to exit an Icon program. The way you have already seen is by returning from the procedure `main`. There are two functions that are also used, `exit` and `stop`.

Table 3 Exiting a program

exit()	exit the program with a normal exit status (i.e., tell the operating system every thing is okay).

Table 3 Exiting a program

exit(i)	exit the program and return the value of integer i as the exit status. This is how to tell the operating system things are not okay, but you will have to know how your OS interprets these exit status values to use this.
stop(s1,s2,...,sn)	write out the strings s1 s2 ... sn and exit with an error status. See a further discussion in Section Chapter 8, I/O, on page 93.

2.5 Elementary numbers

2.5.1 Integer literals

You can write an integer literal (constant) as a decimal number, e.g., `25`.

2.5.2 Selected integer operators

Like most other languages, you use

Table 4 Arithmetic operators

operator	precedence	meaning
+	8	add
-	8	subtract
*	9	multiply
/	9	divide
%	9	remainder
^	10 (right associative)	exponentiation

The operators are executed left to right except for exponentiation which is executed rightmost first. Operators `*`, `/`, and `%` are done before `+` and `-`. Operator `^` is done before any of the others. That is to say, the higher precedence operators are executed before the lower precedence operators.

2.6 Elementary strings

2.6.1 String literals

You write a string literal (constant) surrounded by quotation marks:

```
"Like this"
```

If you need to include a quote in a string, put a backslash in front of it, e.g., `"\""`. If you need to include a backslash, put a backslash in front of it, `"\\"`. There are special ways to include other characters, but we will not discuss them until Chapter 6 on page 61.

2.6.2 Selected string operators

You can concatenate two strings with the || operator, e.g.,

```
s:="ab"
s:=s||"cd"
write(s)
```

will write out `"abcd"`.

You can find out the length of a string using the unary `*` operator, *e.g.*,

```
s:="abc"
write(*s)
s:="a"
write(*s)
s:=""
write(*s)
```

will write out

```
3
1
0
```

2.6.3 Subscripting strings

The characters in a string are numbered from 1 through the length of the string. You can subscript a string the same way you subscript an array in most other languages, put the index in brackets following the string:

```
s:="find"
write(s[3])
s[4] := "e"
write(s)
```

will write out

```
n
fine
```

Unlike most other languages, there are no individual character values. There are only character strings of length one. Expression `s[3]` above returned a length one string, `"n"`.

When you assign to a subscripted string, you can assign more or fewer than one character. For example,

```
s:="fund"
s[4] := ""
write(s)
s[3] := "nny"
write(s)
```


will write

```
fun
funny
```

Icon also allows you to subscript a string with a range of positions, selecting more or fewer than one character. You use the form

```
s[i : j]
```

(where $i \le j$) which selects the substring from character *i* up to, *but not including*, character *j*. (The actual rule, as we will see in section 6.2 on page 62, is that the string positions are between the letters and at each end, so s[i] refers to the letter to the right of position i, and s[i:j] refers to the characters between positions i and j.)

If you assign to the substring, you replace the selected characters. If $i = j$ when you assign, you insert before character *i*. If $i = j = *s+1$, you append to *s*. For example

```
s:="abcd"
write(s)
s[3:3]:="x"
write(s)
s[*s+1:*s+1]:="yz"
write(s)
```

writes out

```
abcd
abxcd
abxcdyz
```

To recapitulate the rules, where s is a string variable and $1 \le i \le j \le *s+1$:

s [i] selects the single character substring, character i, $i \le *s$.

s[i:j] (where $i \le j$) selects the substring from character i through character j-1. If i=j, the substring is empty, but it is a particular substring at a particular position in string `s`, which is important when you assign to it.

You can assign a string of any length to s[i] or s[i:j]. The assignment

```
s[i]:=t
```

behaves like:

```
s:=s[1:i] || t || s[i+1:*s+1]
```

and

```
s[i:j]:=t
```

behaves like:


```
s:=s[1:i] || t || s[j:*s+1]
```

2.6.4 Comparison operators

These are the elementary comparison operators for numbers and strings:

Table 5 Comparison operators

numeric comparison	string comparison	precedence	will succeed when the operands are
i = j	s1 == s2	6	equal
i ~= j	s1 ~== s2	6	not equal
i < j	s1 << s2	6	less than
i <= j	s1 <<= s2	6	less than or equal
i > j	s1 >> s2	6	greater than
i >= j	s1 >>= s2	6	greater than or equal

2.7 Elementary control constructs

Here are the three most common control constructs used in Icon. We are omitting most of the details until Chapter 4 on page 45:

2.7.1 If expressions

You can choose what code to execute using the if expression:

```
if expr1 then expr2 else expr3
```

For the moment we will just use a single comparison operator in *expr1*. If *expr1* succeeds (in other languages we would say, *if expr1 is true*, but we do not say that in Icon), then Icon executes *expr2*, otherwise, if *expr1* fails, Icon executes *expr3*.

Chapter 3 on page 35 and Chapter 4 on page 45 will discuss the other options for *expr1* in much greater detail.

Because the `if` expression is an expression, it returns a value, the value of either *expr2* or *expr3*.

2.7.2 While expressions

You can use the while expression to execute some code repeatedly:

```
while expr1 do expr2
```

Again, for the moment, we will restrict ourselves to a single relational operator in *expr1*. As long as *expr1* succeeds, Icon executes *expr2*. Even though the `while` expression is an expression, it does not return a value.

2.7.3 Expression sequences

You can use braces, `{ expr1; expr2;...; exprn }`, to group sequences of expressions to include them in an if expression or a while expression. The expressions within the braces are separated by semicolons, or by new lines, or both, just like the expression sequence in the body of a procedure.

The expression sequence is an expression. It returns the value of the last expression in the sequence.

2.8 Elementary generators

Generators, expressions that deliver a sequence of values, are the heart of Icon. They will be covered in depth in Chapter 3 on page 35. Here we just show one of the uses.

The loop

```
every i := 1 to 10 do e
```

is the Icon equivalent of a for loop. The expression `1 to 10` is a generator that generates the integers 1, 2, ..., 10. Each of the values is assigned to variable *i* and the expression *e* is evaluated.

Several generators can be combined with an & operator to give the effect of nested loops:

```
every i := 1 to 10 & j := 1 to 10 do e
```

behaves like two nested for loops. For i = 1, j will iterate from one to 10, then for i = 2, j will go from 1 to 10, and so on.

You can also put in tests to eliminate some of the iterations:

```
every i := 1 to 10 & j := 1 to 10 & i ~= j do e
```

will omit evaluating e if i and j have the same value.

2.9 Elementary lists

2.9.1 List creation: list(n)

A list is like an array in other languages. You can create a list whose elements are numbered 1, 2, ..., n using the list function,

```
list(n)
```

For example,

```
L:=list(3)
```

will create a list of length 3 and assign it to `L`. All the elements of the list will be initialized to `&null`, the same as variables are.

2.9.2 Subscripting lists

A list of length n is an array of n elements with the elements numbered from 1 through n, just like arrays. You subscript a list the same way as a string: put the subscript expression in brackets following the list, *e.g.*,

```
L:=list(2)
L[1]:=5
L[2]:=10
write(L[1])
```

writes

```
5
```

You can create a list of length zero. Just call `list(0)`.

2.9.3 List creation: [...]

If you want to create a short list with specific values in it, there is no need to put the values in with assignment expressions. You can list the values you want in brackets:

```
L:=[5,10]
write(L[1])
```

writes

```
5
```

You can create a list of length zero by writing `[]`.

2.9.4 List creation: list(n,x)

If you want to create a list with all elements the same, but not `&null`, use `list(n,x)` which will create a list of length `n` all of whose elements are `x`.

2.9.5 Selected list operators

You can concatenate two lists with the **|||** operator, *e.g.*,

```
s:=[5,6]
s:=s|||[7,8]
write(s[3])
```

will write out `7`.

The result of `x|||y` is a new list containing the elements of `x` followed by the elements of `y`. Lists `x` and `y` are not altered.

You can find out the length of a list using the unary ***** operator, *e.g.*,

```
s:=[1,2,3]
write(*s)
```



```
s:=[]
write(*s)
```

will write out

```
3
0
```

You can compare two lists to see if they are the same list or not by using the **===** or **~===** operators. (Use three equal signs in a row.) What does it take to be the same list? Consider

```
L:=[1,2]
M:=L
```

After this code, `L === M` will succeed and `L ~=== M` will fail. The assignment `M:=L` makes `L` and `M` point to the same list. Now consider

```
L:=[1,2]
M:=[1,2]
```

After this code, `L ~=== M` will succeed and `L === M` will fail. The assignment `M:=[1,2]` makes `M` point to a new list which cannot be the same as `L`. Even though `L` and `M` point to lists that have the same length and the same contents, they are not the same.

Example:

```
x:=1
y:=1
write(if x === y then "equal" else "not equal")
```

will write

```
equal
```

Example:

```
x:=1
y:="1"
write(if x === y then "equal" else "not equal")
```

will write

```
not equal
```

Example:

```
L:=[[1],[1]]
write(if L[1] === L[2] then "equal"
     else "not equal")
```


will write

```
not equal
```

since the two separate occurrences of `[1]` create two different lists.

Example:

```
L:=list(2,[1])
write(if L[1] === L[2] then "equal"
     else "not equal")
```

will write

```
equal
```

since the occurrence of `[1]` in the list function is evaluated just once, creating one list, which is assigned to both elements of `L`.

2.9.6 Differences between lists and strings

Here are some differences between lists and strings:

There are string literals. There are no list literals.

Lists are mutable values; strings are immutable. This means you can change an element of a list and see that change in all the lists equal (`===`) to it. If you change a character in a string variable, Icon actually creates a new string with the change made and assigns that new string back to the variable.

A subscripted string is a string of length one. A subscripted list is not usually a list of length one: it's whatever value of whatever type was in that element.

When you assign a value to an element of a list, the length of the list does not change. When you assign a string to a subscripted string, the length of the string can change. The string you assign is spliced in replacing the character at that position.

You can assign any kind of value to any element of a list. You can only assign strings to subscripted string variables.

Procedure `write` will write out a string. It will not write out a list.

You can assign a list to an element of another list. You can assign a list to an element of itself, getting a circular structure.

You can only assign to a subscripted string variable. You cannot assign to a subscripted string constant, *e.g.*,

```
s:="abcd"
s[2]:="x"
```

but not


```
"abcd"[2]:="x"
```

You can always assign to a subscripted list, e.g.,

```
[1,2,3][2]:=5
```

2.9.7 Procedure main's parameter

Procedure main takes one parameter, a list of all the command line arguments as strings. As you have seen above, you do not actually have to declare procedure main with a parameter. Here's an example of using the parameter, a program to echo the command line arguments:

```
procedure main(args)
i := 1
while i <= *args do {
     writes(args[i]," ")
     i:=i+1
}
write() #terminate line
end
```

2.10 Records

You can create new record data types in Icon, just as you can in Pascal (records), C (structs), and C++ (classes). A record type is declared:

```
record rname(f1,f2,...,fn)
```

where

- rname is the name being given to the record type.
- f1, f2, ..., fn are the names being given to the fields (members) of the record.
- the record declaration is only permitted *outside* of procedure declarations.

For example

```
record Point(x,y)
```

might be used to define a point in a two-dimensional coordinate system.

A point may be created by the expression:

```
r := Point(1,2)
```

which will create a new record of type *Point*, initialize its *x* field to 1 and its *y* field to 2, and assign the point record to variable *r*.

The fields of the record can be accessed using the binary ".", field referencing, operator, e.g.

```
r.x := r.y
```


Unlike Pascal, C, and C++, there are no pointers and no distinction between accessing the field of a record variable (e.g. `r.f1` in C) and accessing the field of a record via a pointer (`r->f1`). Internally, Icon accesses all records through pointers.

```
p1 := Point(1,2)
p2 := p1  #p2 points to p1
p2.x := 2 #also changes p1.x
write(if p1 === p2 then "equal" else "not equal")
write(if p1.x === p2.x then "equal"
     else "not equal")
```

will write

```
equal
equal
```


Chapter 3 Generators

3.1 Expressions are generators

The greatest difference between Icon and other programming languages is this: in Icon, expressions are generators. Expressions generate sequences of values.

To be sure, constants and variables generate only single values, but there are language constructs that generate more than one value, and there are some that may generate none.

Expressions generate their values by backtracking. To understand how this works, you will need to understand the order in which expressions are evaluated.

3.2 Expression evaluation order

Generally, expressions are evaluated left to right and bottom up.

- Control goes left to right through an expression evaluating it.
- As soon as the subexpressions are evaluated, the expression that contains them is evaluated.

Consider the expression:

(a-b)+(c-d)*(e-f)

The expression will be evaluated in the order:

1. t_1:=a-b The t_i's are temporary variables.
2. t_2:=c-d
3. t_3:=e-f
4. t_4:=t_2*t_3 now the operands of the multiply are available
5. t_5:=t_1+t_4 now the operands of the add are available

The difference between Icon and more common programming languages is that in Icon, after moving forward through the expression, control can

- back up through the list of operations to find an operation that can generate

more values,

- generate another value from it, and then
- move forward again evaluating the subsequent operations again with the new value.

3.3 Every

One context in which more than one value is required is the `every` expression. The expression

```
every  e1
```

creates a context in which e1 generates all its values.

3.4 To

Some operators can generate a series of values. Consider the operator **to**. The expression

```
1 to 5
```

can generate the series values 1, 2, 3, 4, 5.

Whether it will be allowed to generate all those values depends on the context in which it occurs. Some contexts allow at most one value to be generated. In the sequence of expressions in the body of a procedure, for example, each expression is allowed to generate at most one value. So the following code will write out the single value, 10.

```
procedure main()
write(10 to 20)
end
```

However, the following code will write out the line

```
1 2 3 4 5
```

Figure 6 *Write 1 2 3 4 5*

```
procedure main()
every writes(" ",1 to 5)
write()
end
```

The *e1* `to` *e2* generator steps by `1` from `e1` through `e2` which must be greater than or equal to e1 to generate any values.

The **to** operator associates to the left, *e.g.*,

```
every writes(" ",1 to 2 to 3)
```

writes out


```
1 2 3 2 3
```

The **to** operator doesn't work with large integers, i.e. integers larger than a machine word.

3.5 To-by

There is fuller form of the `to` generator, *`e1 to e2 by e3`*, that will step by the value of *`e3`*. In this case, *`e3`* can be negative, but then `e2` must be less than or equal to `e1` to get any values.

The **to-by** operator doesn't work with large integers.

3.6 Element generation: !e

The `!` operator generates the components of data objects.

```
! x
```

does the following:

- if *x* is a string, *!x* generates the one character substrings of *x* in order from 1, *i.e.*, `x[1 to *x]`
- if *x* is a string variable, *!x* yields variables which can be assigned to.
- if x is a string value, *e.g.*, a literal, !x generates string values and cannot be assigned to.
- if *x* is a list, *!x* generates the elements of *x* in order from 1, *i.e.*, `x[1 to *x]`.
- if *x* is a list, *!x* generates variables, e.g. you can use !x :=0 to assign zeros to each element of a list.
- *!* applies to other data types as well, as will be discussed in sections on those types.

3.7 Backtracking

How does *every* work? Consider the sequence of operations for the expression in the *every* line of Figure 6 on page 36:

1. get the variable `writes`
2. get the string `" "`
3. get the number `1`
4. get the number `5`
5. start up the generator `1 to 5` and generate the first value, `1`.
6. Whenever it resumes, generate the next value from the generator.

7. call the procedure writes passing it the results of steps 2 and 6.

Control goes down this list doing each operation in turn. When it gets to the end of the list, the every forces it to move back through the list looking for a generator. It finds the generator in step 6 and ***resumes*** executing it. The generator generates another value, 2, so control now starts forwards through the code again. Eventually, once 5 has been written out, the generator will not be able to give another value. So control will continue searching back looking for another generator. Not finding one, it will fall off the front of the list, which tells control it is done evaluating the expression.

This process of searching back through the sequence of operations looking for a generator is called ***backtracking***.

3.8 Failure

Some control constructs, like the every-expression, cause backtracking into expressions.

Another way to cause backtracking is expression ***failure***. To anthropomorphize a bit, if an operation ***fails***, control backtracks to see if it can generate some values that will make the operation succeed.

There is a built-in expression that always fails: **&fail**. Whenever Icon executes it, it always backs up. The effect of the every-expression in could be achieved by:

```
writes(" ",1 to 5) + &fail
```

The `&fail` causes control to keep backtracking to the `1 to 5` generator, generating a new number and writing it out. What does the `+` do? Nothing. The `&fail` prevents control from ever reaching the `+`. It is never executed.

3.9 Binary operators containing generators

Normal binary operators, like the arithmetic operators, simply compute a single output value for a pair of operands. They only generate more values when generators earlier in the list of operations cause them to be reevaluated.

If both subexpressions have generators in them, then the generator in the right subexpression runs to completion before the generator in the left subexpression is allowed to generate another value. Once the left generator has generated a new value, the right generator is reinitialized and starts anew. For example,

```
every writes(" ",(1 to 2) + (10 to 20 by 10))
```

writes 11 21 12 22

3.10 Arithmetic relational operators

A relational operator is a binary operator that succeeds if the relation holds be-

tween its operands and fails if it does not. For example,

`a < b` will succeed if a's numeric value is less than b's and fail if a's value is greater than or equal to b's. (It will stop the execution of the program if the operands are not numeric and cannot be converted to numeric.)

The relational operator yields the value of its right operand if it succeeds. Since relational operators are performed left to right, Icon allows such convenient forms as

`a < b < c` which succeeds if a is less than b and b is less than c, yielding the value of c, and fails otherwise

One common trick in Icon is to write conditional assignment statements. Suppose you want to assign to variable x the maximum of the values of x and y. You can write:

```
x:= x < y
```

This says:

- If x is less than y then assign the value of y to x. Since y is the maximum of x and y, this is what we want.
- If however x is equal to or greater than y then the relation fails and the overall assignment expression stops executing before the assignment is done. Since x has the maximum value, we do not want to assign it a new value.

3.11 Conjunction: e1 & e2

There is a special binary operator, &, read "and," that simply returns the value of its right operand. It has the lowest precedence, even lower than assignment.

The & operator is used for control rather than computation. It allows you to write a sequence of generators and tests. For example,

```
every i:=1 to 3 & j:=1 to 3 & i~= j &
   writes(" ",i+j)
```

writes out

```
 3 4 3 5 4 5
```

3.12 Null and non-null tests: / x and \ x

There are two unary operators that compare their operands to &null and succeed

or fail if they are equal.

Table 6 Testing for &null

`/ x`	succeeds if `x` has the value &null and fails otherwise
`\ x`	succeeds if `x` does not have the value &null and fails otherwise

If these operators are given variables as operands, they leave them as variables. Therefore, they can be used for conditional initialization. For example

```
/ x := 0
```

will assign x the value 0 if x had the value &null. Since variables are initialized by the system to &null, this can be used to initialize a variable the first time it is encountered.

Another use is to represent Boolean values. Just allow &null to represent false and anything else to represent true. What you might write in some other language as “if (x and y) then...” you would write in Icon as “if \x & \y then...”.

3.13 Coevaluation

A list of expressions separated by commas within a pair of parentheses are evaluated as if they were separated by &’s. For example

```
every writes(" ",(1 to 3 , 1 to 2))
```

behaves the same as

```
every writes(" ",1 to 3 & 1 to 2)
```

This is called *co-evaluation*.

Generally

```
( e1, e2, ..., en)
```

is equivalent to

```
( e1 & e2 & ... & en)
```

Why are there two ways to do the same thing? This is probably because co-evaluation is a degenerate version of *selection*. If you place an integer, k, in front of the parentheses, i.e.,

```
k (e1, e2, ..., ek, ..., en)
```

it will return the value of the kth expression. When used as a generator, for every combination of values the expressions generate, this form will return the kth.

So,

```
( e1, e2, ..., en )
```


is equivalent to

```
n (e1, e2, ..., en)
```

For example,

```
every writes(" ",(1 to 3, 1 to 2))
```

writes

```
 1 2 1 2 1 2
```

while

```
every writes(" ",1 (1 to 3, 1 to 2))
```

writes

```
 1 1 2 2 3 3
```

In fact, the general form of selection is

```
e0 (e1, e2, ..., ek, ..., en)
```

where the value of expression *e0* selects the value to return. Naturally, *e0* is evaluated before the other expressions and can be a generator.

```
every writes(" ",(1 to 2) (1 to 3, 1 to 2))
```

writes

```
 1 1 2 2 3 3 1 2 1 2 1 2
```

If *e0* returns a procedure rather than a number, this syntax does not mean selection: it is a procedure call. See Section 5.1, Procedure calls, on page 53.

3.14 Alternation: e1 | e2

The vertical bar, `|`, read “or,” looks like a binary operator but does not behave like one. Expression

```
e1 | e2
```

generates all values generated by `e1` followed by all values generated by `e2`.

```
1 | 2 | 3 | 4
```

generates the sequence 1, 2, 3, 4.

So does

```
(1 | 2) | (3 | 4)
```

The `|` operator has lower precedence than comparison operators, so you could write


```
i = j | i = k
```

to see if i is equal to either j or k. There is, however, a different idiom for that:

```
i = (j | k)
```

You can use | on the left hand side of an assignment to generate variables, e.g.,

```
every a | b | c := 0
```

although it is neither as clear nor as efficient as

```
a := b := c := 0
```

3.15 Sequence generation: seq(...)

The `to` and `to-by` operators put a limit on the number of values generated. If you do not want to put a limit, you can use the system function `seq` to generate a sequence of values.

Table 7 Function seq( )

`seq()`	generates the sequence 1,2,3,...
`seq(i)`	generates the sequence i,i+1,i+2,...
`seq(i,j)`	generates the sequence i,i+j,i+2j,...; j must not be 0.

The **`seq`** operator doesn't work with large integers.

3.16 Repeated alternation: | e

A generator that is quite difficult to use well is the unary vertical bar, `|`. If you put `|` in front of a generator, `g`,

```
| g
```

it will allow `g` to generate all its values, and then it will reinitialize `g` and allow it to generate all its values again, and again, and again. Only if `g` immediately fails on some initialization will `| g` fail and allow control to backtrack past it. Here are some examples:

Table 8 Examples of |e

Expression	Generates	Explanation
`| 3`	`3, 3, 3, .....`	The literal 3 generates a single value. The | repeatedly initializes it to generate another value 3.
`|(0 | 1)`	`0, 1, 0, 1, 0, 1,...`	

Table 8 Examples of |e

Expression	Generates	Explanation
`| &fail`		`&fail` fails immediately. Since its operand did not generate any values, the `|` also fails.
`(i:=0) & |(i+:=1)`	`1, 2, 3, ...`	similar to `seq()`, except that it uses the variable `i` to hold the values.

3.17 Limitation: e1 \ e2

If you use `seq()` or `|e` , you risk creating infinite computations. One way to prevent that is the limit operator:

```
e1 \ e2
```

will allow `e1` to generate at most `e2` values. For example,

Table 9 Examples of e1\e2

expression	generates	Explanation
`seq()\5`	`1, 2, 3, 4, 5`	
`(1 to 3)\5`	`1, 2, 3`	
`(1 to 9)\5`	`1, 2, 3, 4, 5`	
`1 to 9\5`	`1, 2, 3, 4, 5, 6, 7, 8, 9`	The \5 limits the number of values generated by the literal 9. The limit operator has a very high precedence, just lower than unary operators.
`|1\5`	`1, 1, 1, 1, 1`	
`|(1 to 3) \ 5`	`1, 2, 3, 1, 2`	

Table 9 Examples of e1\e2

expression	generates	Explanation
`(1 to 3)\(2 to 4)`	`1 2 1 2 3 1 2 3`	Unlike all other operators, the limit operator evaluates its right operand before its left. For every value generated by its right operand, it generates and limits the number of values generated by its left.

As the example `(1 to 3)\(2 to 4)` shows, the limit operator evaluates its operands right before left. **Note: This is the only exception to Icon's left-to-right evaluation order.**

3.18 Idiom: generate and test

We know by the Pythagorean Theorem that the sum of the squares of the lengths of the sides of a right triangle is equal to the square of the length of the hypotenuse. Suppose we want to find all lengths of sides of right triangles that are integers in the range 1 through 100 (including the hypotenuse). We can do it with the following program:

Figure 7 Figure 2 Right triangles

```
procedure main()
every i:=1 to 100 &
     j:=i to 100 &
     m:=j to 100 &
     m*m = i*i+j*j &
          write(i," ",j, " ",m)
end
```

This is an example of **generate and test** paradigm often used in artificial intelligence and combinatoric search programs. The goal here is specified by the test `m*m = i*i+j*j`. Generators preceding the tests will try to find some values that will pass the tests.

Chapter 4 Control Constructs

4.1 {e1; e2; ; en }

Braces are used to surround a sequence of expressions. Every expression except the last is evaluated in sequence to produce at most one value. Whether it succeeds or fails, control then goes on and evaluates the next expression in the sequence. The last expression, en, will then generate as many values as the surrounding context allows. For example,

```
every {writes(" ",1 to 5);writes(" ",6 to 10)}
```

writes out

```
1 6 7 8 9 10
```

The semicolon can be replaced with a new line in the sequence, just as in the expression sequence in the body of a procedure.

4.2 every do

We have already seen the simple form of an every expression:

```
every e1
```

which generates all the values for *e1*.

Another form is

```
every e1 do e2
```

which says: for each value of *e1*, evaluate *e2*. Icon does the following:

- It will cause `e1` to generate all the values it can.
- Each time `e1` generates a value, Icon evaluates `e2`.
- Expression `e2` may succeed or fail.
- Icon will generate at most one value for `e2` each time it evaluates it.
- The every-do expression fails when `e1` fails.

So the expression `every` *e1* `do` *e2* behaves very much like `every` *e1* `&` `(`*e2*`)\1`. However, it looks more like loops in conventional languages with the `do`, so it's more intuitive. It behaves differently with respect to the `break` and

`next` expressions, discussed in section 4.11 and section 4.12 on page 51.

Every-expressions are expressions and can be placed inside other expressions. What values do they generate? Actually, none. Like `&fail`, they fail without returning a value. However, if they are exited by a break expression, they can return values. See Section 4.11, break, on page 51.

4.3 if then else

```
if e1 then e2 else e3
```

- The if-expression behaves much like the if-statement in conventional languages with these differences:
- The expression *e1* is evaluated to determine its success or failure. If *e1* succeeds, the if expression behaves like *e2*; if *e1* fails, it behaves like *e3*.
- Expression *e1* is a ***bounded expression***—at most one value is generated from *e1*. Control is cut off from backing into *e1* after it has generated one value.
- Either *e2* or *e3* may generate as many values as the context requires.

As in other languages, there is a version that omits the else expression.

```
if e1 then e2
```

If the **else** clause is omitted and *e1* fails, the entire **if** expression fails.

For example, to sort a pair of numbers in variables x and y so that the smaller is in x and the larger in y, you can write:

```
if x>y then x:=:y
```

4.4 idiom : goal directed evaluation

Consider the following program to find the lengths of the sides of some right triangle for which the sum of the sides is no less than 1000 and no more than 2000 units long:

Figure 8 Find a right triangle

```
procedure main()
local i,j,m,n
if i:=1 to 2000 &
     j:=i to 2000 &
     n:=i*i+j*j &
     m:=integer(sqrt(n)) &
     m*m = n &
     1000 <= i+j+m <= 2000
     then
          write(i," ",j," ",m)
end
```

In other programming languages, you would probably have to write several

nested loops to find such a triangle. (By the way, the one it finds has sides 33, 544, and 545.) Once you've found it, you'd have to jump out of the nest of loops to go on with the program.

This is an example of goal directed evaluation. The goal is represented by the tests `m*m = n & 1000 <= i+j+m <= 2000`. The computation can be viewed as trying to generate sides i and j that achieve the goal.

4.5 case of { }

The case expression behaves similarly to the case or switch statement in other programming languages. The general form

```
case e1 of {
e2 : e3
e4 : e5
...
e2n : e2n+1
default : e2n+2
}
```

is equivalent to

```
{(tmp := e1 )\1 &
if tmp===(e2) then e3
else if tmp===(e4) then e5
...
else if tmp===(e2n) then e2n+1
else e2n+2
}
```

where `tmp` is an otherwise unused variable. To express its behavior another way:

- the expression e_1 is evaluated to give one value which is assigned to a temporary variable.
- if the expression e_1 fails, the case expression fails.
- in order, for i from 1 to n, Icon compares the value of expression e_1 to the values generated by expression e_{2i} .
- the comparison operator for the equality comparison is the universal equality test, ===. If it is given two numbers, it performs a numeric test; if two strings, a string test. For other kinds of objects, it tests to see whether they are the same object or not.

- as soon as Icon finds a value equal to some value generated by an e_{2i} , it evaluates e_{2n+1} to yield the values of the case expression.
- if none of the expressions e_{2i} generate a value equal to the value obtained from e_1, then the default expression, e_{2n+2}, is evaluated as the value of the case expression.
- the default expression, e_{2n+2}, is optional. If it is not present and no other expression is selected, the case expression fails.

4.6 while do

```
while e1 do e2
```

The while expression is like the while statement in other programming languages. It differs from the every expression in that it generates at most one value from *e1* per iteration. Briefly:

- it evaluates *e1*
- if *e1* succeeds, it evaluates *e2*
- *e2* may succeed or fail
- it generates at most one value from *e1* and at most one value from *e2* each iteration
- once *e2* has been evaluated, it restarts evaluation of *e1* from the beginning
- it fails when *e1* fails

There is a version without the do clause:

```
while e
```

Which repeatedly evaluates *e* from the beginning until it fails.

For example, here is a file copy program. It copies its input into its output. If you are running it from the keyboard, it writes back each line you type to it.

Figure 9 *File copy*

```
procedure main( )
while write(read( ))
end
```

The `read` procedure reads one line of input at a time and will fail when the input is finished.

The file copy cannot use an **every** expression. The `read` procedure is not a gen-

erator. It will return one line each time it is entered as control moves forward, but it cannot be resumed during backtracking; backtracking will back past it.

Here is a small utility program to create define macros in C for bit positions. You feed it a list of identifiers and it will create one `#define` line for each identifier. This makes each one a different power of two. That is, it will make each one a mask for a different bit position.

Figure 10 *Define bit positions*

```
procedure main()
i := 1
while x := read() do {
     write("#define ",x," ",i)
     i := i+i
}
end
```

4.7 not

```
not e
```

The unary operator `not` *e* will succeed if its operand *e* fails and fail if it succeeds. If `not` succeeds, it returns `&null` (It has to return something, and its operand did not give it any value to return.)

Since not is a unary operator, it has higher precedence than any binary operator. You will almost always need to surround its operand with parentheses.

4.8 idiom: write "all do" as "not any don't"

Icon evaluates expressions to try to find some way to make them succeed. Success means there exists at least one way to make it succeed. For example, suppose we want to know if there are two elements of a list, L, that are equal. We could write:

```
i:=1 to *L-1 & j:=i+1 to *L & L[i]===L[j]
```

Sometimes you want to know whether something's true for *all* rather than whether it's true for *any*. There is no way to write that directly in Icon. You have to resort to a double negative.

Something is true for all if and only if there are none for which it is not true.

Suppose we wish to find whether all elements of a list are equal. If all are equal to each other, they are equal to the first element, so if there is any element not equal to the first, then they are not all equal. We could write the test as

```
not (i:=2 to *L & L[1]~===L[i])
```


4.9 until do

```
until e1 do e2
```

The until expression behaves like

```
while not ( e1 ) do e2
```

Briefly:

- it evaluates *e1*
- if *e1* fails, it evaluates *e2*
- *e2* may succeed or fail
- it generates at most one value from *e2* each iteration
- once *e2* has been evaluated, it restarts evaluation of *e1* from the beginning
- it generates at most one value from *e1* for it fails as soon as *e1* succeeds

There is a version without the do clause:

```
until e
```

Which repeatedly evaluates *e* from the beginning until it succeeds.

For example, consider the following program to generate Fibonacci numbers up to 100,000. Fibonacci numbers come in a sequence defined as follows: the first two Fibonacci numbers are 1; each subsequent number is the sum of the two previous Fibonacci numbers.

Figure 11 Write Fibonacci numbers using until

```
procedure main()
local i,j
i:=1
j:=1
until i>100000 do {
     write(i)
     i+:=j
     i:=:j
}
end
```

4.10 repeat

The repeat expression

```
repeat  e
```

will repeatedly evaluate the contained expression *e*. It will never terminate unless exited explicitly by, for example, `break`, `return`, `suspend`, or `fail`.

4.11 break

The `break` expression is used to exit from a surrounding loop of any type (`every`, `while`, `until`, `repeat`). There are two forms, break alone, or break with a contained expression:

```
break
break expression
```

For the simple `break`, control exits the immediately surrounding loop. Unlike falling out of the loop, which fails, exiting with a `break` succeeds yielding the value of the *expression*. If the *expression* is absent, it yields a `&null`.

You could write something like

```
if every {... break ...} then exited_by_break()
else exited_normally()
```

The form `break` *expression* exits the loop and generates the values the *expression* generates. In fact, the *expression* behaves as if it is outside the loop. If it contains a `break` or `next`, they apply to the next surrounding loop.

To exit two levels of loop, use

```
break break
```

Fibonacci numbers, again:

Figure 12 Fibonacci using repeat

```
procedure main()
local i,j
i:=1
j:=1
repeat {
     write(i)
     i+:=j
     i:=:j
     if i>100000 then break
}
end
```

4.12 next

```
next
```

The expression `next` causes the loop it's in to start its next iteration.

If it's within an every expression,

```
every e1 do e2
```


it immediately causes backtracking back into the control expression, *e1*.

If it's within a while or until,

```
while e1 do e2
```

or

```
until e1 do e2
```

it enters the control expression, *e1*, again.

If it's within a repeat,

```
repeat e1
```

it just starts evaluating *e1* again.

Chapter 5 Procedures

5.1 Procedure calls

A procedure call has the form:

$expr_0$ ($expr_1$, $expr_2$, ..., $expr_n$)

- Expression $expr_0$ evaluates to a procedure value.
- Most commonly, $expr_0$ is a global name that has been assigned the procedure value either by a procedure declaration or by being the name of a built-in function.
- Expression $expr_0$ may be an expression that evaluates to or generates procedures, e.g., `(F|G)(x,y)` would call `F(x,y)` and then, if backtracked into, call `G(x,y)`.
- The actual parameter list, $expr_1$, $expr_2$, ..., $expr_n$, may be empty.
- The number of actual parameters provided may be more or less than the number of parameters specified in the procedure declaration. Extra parameters are ignored. Missing parameters are given the value `&null`.
- Expressions may be omitted from within the actual parameter list, e.g.

  ```
  F ( , 2, , 4).
  ```

  The missing parameters are given the value `&null`.
- The parameters are passed by value. If there is a variable in the parameter list, its value is passed to the procedure.

  However, Icon waits until the moment of call to fetch the current value of a variable. For example

  ```
  write(x:=1," ",x+:=1," ",x+:=1)
  ```

  writes

  ```
  3 3 3
  ```

  since each assignment returns the *variable* on the left hand side (`x`) and the write fetches the current value of each of the occurrences of `x` just before the call and after the final assignment has been done.
- The parameters are evaluated left to right. All must succeed giving a value before the procedure is called.

- The procedure is called for each set of values the parameters generate as long as control backs into it.
- The procedure itself may *fail*, may *return* a value, or may *generate* a sequence of values.
- The procedure may return or generate variables that can be assigned to.

5.2 Procedure declarations

The form of a procedure declaration is

```
procedure name ( formal1, formal2,..., formaln )
declarations
initial_expression_option
 expression_sequence
end
```

- The procedure declaration creates a procedure value.
- The name of the procedure becomes a global variable initialized to the procedure value.
- The procedure value can be assigned to other variables and put in data structures.
- The list $formal_1$, $formal_2$,..., $formal_n$ is a list of zero or more identifiers separated by commas. They are the formal parameters of the procedure.
- Each formal parameter is assigned the value of the corresponding actual parameter. Since mutable objects are accessed by pointer, the pointer is copied to the formal parameter and both the formal and the actual parameters point to the same object, which resembles call-by-reference.
- If there are more actual parameters than formal, the remaining actual parameters are ignored.
- If there are fewer actual parameters than formal, then the extra formal parameters are assigned the initial value `&null`.
- You may write a procedure to take a variable number of parameters by following the *last* formal parameter, $formal_n$, with a pair of brackets, `[]`. All the actual parameters from n on will be placed in a list which will be assigned to $formal_n$. (Lists are discussed in Chapter 9 on page 105.)
- The declarations allowed are `local` and `static`. See Section 2.3, Declarations, on page 23.
- The `initial` expression is optional. It is discussed in section 5.7 on page 57
- The *expression_sequence* is like the expression sequence between braces, `{...}`—the expressions are evaluated in sequence, each to produce a sin-

gle value or to fail. Each expression is *bounded*: control will *not* backtrack into an expression once it has produced a value.

- If control finishes executing the expression sequence and comes to the `end`, the procedure call *fails*.

Figure 13 Bit positions in octal

```
procedure oct(i)
return if i = 0 then "0" else oct(i / 8) || i % 8
end
procedure main()
i := 1
while x:= read() do {
write("#define ", x, " ", oct(i) )
i +:= i
}
end
```

5.3 Idiom: default values for parameters

You can assign default values to parameters as follows:

```
/ parameter_name := default_value
```

If the caller did not provide an actual parameter, Icon supplies `&null`. The / tests the value of the parameter and succeeds only if it is &null. The &null is replaced with the default value.

5.4 Return

A procedure returns a value by executing a return expression.

```
return e
```

or

```
return
```

The return behaves as follows:

- If the `return` does not include an expression *e*, the procedure returns `&null`.
- If the `return` contains an expression, *e*, the procedure returns the value of *e* to the caller.
- If the `return` contains an expression, *e*, and expression *e* fails, however, the procedure call fails causing backtracking in the caller.
- The `return` does *not* create a generator. The procedure does not suspend. It cannot be reentered to generate another value. At most one value is returned.

- If *e* is a variable, but not a variable declared `local` or `static`, the variable is returned from the procedure. That is, you can potentially assign a value to what a procedure returns. You can use a procedure call on the left hand side of a `:=`. A local variable cannot be returned because it no longer exists after the procedure returns.

Consider the following procedure that will return the maximum of two numbers:

Figure 14 Maximum of two numbers

```
procedure max2(x,y)
x <:= y
return x
end
```

5.5 Fail

The fail expression

```
fail
```

causes the call of this procedure to fail. The `fail` will cause backtracking in the caller.

5.6 Suspend

You use `suspend` to make a procedure into a generator. The `suspend` behaves like `return`, except that it leaves the procedure as a generator, ready to be resumed to try to generate more values. The forms are:

```
suspend  e1 do e2
suspend  e1
suspend
```

- The `suspend` has the following behavior:
- The `suspend` evaluates its contained expression, *e1*, and passes each value *e1* generates back to the caller. It is like a `return`, but the procedure doesn't go away after returning one value.
- When the caller backs into the procedure call, control backs to the procedure, executes *e2* if present, and backs into the expression *e1* to generate the next value.
- When expression *e1* fails to generate any more values, control falls out of the `suspend` exactly like control falls out of an `every`. If it fails to generate any values, control moves on without delivering any values to the caller for this `suspend`.
- Without the included expression *e1*, `suspend` delivers `&null` back to the caller.

- If *e1* is a variable, but not a variable declared `local`, the variable is returned from the procedure. That is to say, you can potentially assign a value to what a procedure generates. You can use a procedure call on the left hand side of a `:=` .

For example,

Figure 15 Demonstration of suspend

```
procedure G()
suspend |writes(" e1")\3 do writes(" e2")
write()
suspend |writes(" e3")\2
write()
end

procedure main()
every G() do writes(" e4")
end
```

writes out

```
 e1 e4 e2 e1 e4 e2 e1 e4 e2
 e3 e4 e3 e4
```

5.7 Initial

The `static` declaration declares variables within a procedure that retain their values between calls. The problem, of course, arises the first time the procedure is called: the variable has no value from before. Icon provides a way around this problem with the initial declaration:

```
initial e
```

which comes just after the declarations at the start of the procedure and just before the expression sequence that is the body of the procedure.

The contained expression *e* is executed only once, the first time the procedure is called.

The initial expression is used to initialize the static variables declared within the procedure. It is also used to initialize some global variables shared by a collection of procedures.

Suppose you have two procedures, `enqueue(x)` and `dequeue()` that are supposed to put items into and remove them from a shared queue implemented as a list. They need that list initialized, so they might be written (omitting the actual code):

Figure 16 Using initial

```
global queue
```



```
procedure enqueue(x)
initial /queue:=list()
...
end
procedure dequeue()
initial /queue:=list()
...
end
```

Each of them checks to see if the queue has been initialized yet and initializes it if it has not been. By using the `initial` expression, they avoid having to check each time

5.8 String invocation

You can call a procedure by supplying its name in a string. For example, if you have declared a procedure `f`, you can call it

```
"f"(x)
```

rather than

```
f(x)
```

However, recently Icon has been optimized to save space, so you have to tell it to keep around the names of the functions you wish to call via their names. You can do it in either of two ways:

1) translate it with the command

```
icont -fs ...
```

where the flag -fs says to implement strings fully, including keeping the names of functions.

2) include `invocable` commands for each procedure you may call by its string name, *e.g.*,

```
invocable "f"
```

at the same level in the program as `global` declarations, *i.e.*, outside procedures. Be sure to enclose the procedure name in quotation marks.

Or to make all procedures invocable, use

```
invocable all
```

5.9 Applying a procedure to a list

Suppose you want to apply a procedure, `P`, to arguments that are contained in a list, `L`. Use the infix `!` operator:

```
P ! L
```


Operator `!` takes a procedure on its left hand side and a list on its right hand side and calls the procedure passing the arguments contained in the list. For example, the following all call procedure P passing it a and b:

P(a,b)	P ! [a,b]	"P"(a,b)	"P"![a,b]

5.10 Functions that apply to procedures

There are several built-in functions that work with procedure objects. The function `args` reports the number of arguments that a procedure requires. The function `proc` will return the procedure named by a string. This may not seem useful, given string invocation of procedures, but it will also give procedures corresponding to operators and allow you to choose between unary and binary or binary and ternary. In addition, there are two useful procedures in the Icon Program Library. Procedure `prockind` will tell you what kind of procedure a procedure object is. Procedure `procname` will return the name of a procedure.

Table 10 Procedures that apply to procedures

`args(p)`	returns the number of parameters required by procedure p. If p is a user procedure with a variable number of parameters, `args(p)` returns the negative of the number of parameters p was declared with. If p is a built-in procedure with a variable number of parameters, `args(p)` returns -1.
`proc(s)`	returns the procedure named s, where s is a string.
`proc(s,i)`	returns the procedure for the operator whose name is s which takes i parameters, e.g., `proc("*",1)(x)` *x* `proc("*",2)(x,y)` *x*y* `proc("[]",2)(x,y)` *x[y]* `proc("[:]",3)(x,y,z)` *x[y:z]* `proc("...",2)(x,y)` *x to y* `proc("...",3)(x,y,z)` *x to y by z*
`prockind(x)`	fails if `x` is not a procedural value. Otherwise, it returns "`c`", if `x` is a record constructor, "`f`", if `x` is a built-in function, "`o`", if `x` is an operator, or "`p`", if `x` is a user-defined function. `link prockind`

Table 10 Procedures that apply to procedures

`procname(x)`	returns the name of the procedure value `x` (which can also be a record constructor or operator), or fails if `x` is not a procedure. If `x` is an operator, its name has its number of parameters appended on the right, e.g. `procname(write)` yields "`write`" `procname(proc("...",3))` yields "`...3`" `link procname`
`type(p)`	"`procedure`"

Chapter 6 Strings and Character Sets

6.1 String literals

String literals are enclosed in double quotes, ". The backslash character is the incorporation character that is used to include quotes, backslashes and special characters into the string.

Table 11 Representation of special characters

Character sequence	represents
`\b`	backspace
`\d`	delete
`\e`	escape
`\f`	form feed
`\l`	line feed
`\n`	new line
`\r`	return
`\t`	tab (horizontal)
`\v`	vertical tab
`\'`	single quote
`\"`	double quote
`\\`	back slash
`\qqq`	the character with the octal code *qqq*. *qqq* represents up to three octal digits. As long as the following character is not a digit in the range 0 through 7, you may use fewer than three digits.

Table 11 Representation of special characters

Character sequence	represents
`\xhh`	the character with the octal code *hh*. *hh* represents up to two hexadecimal digits, 0 through 9 and A through F (either upper or lower case) representing 10 through 15. As long as the following character is not a hexadecimal digit, you may use one hex digit.
`\^c`	the control code *c*.

If you want to continue a string literal onto another line, break it just before a printing character. Place an underscore at the end of the first line. Continue the literal on the second line, preceded, if you wish, by blanks and tabs For example,

```
"hello, _
     world"
```

6.2 Positions in strings

The discussion in Section 2.6, Elementary strings, on page 25 was a bit simplified. The positions in a string are not the positions *of* characters but the positions *between* characters. The leftmost end of a string is position 1. The rightmost end's position is the length of the string plus one.

In addition, you are permitted to use zero and negative numbers as subscripts. Position 0 is the rightmost end of a string, -1 is the position just before the rightmost character, and minus the length of the string is the position of the left end.

For example, the positions in string "Frog" are:

```
"   F    r    o    g   "
    1   2    3    4   5
   -4  -3   -2   -1   0
```

6.3 Subscripting

If you subscript a string

```
s [ i ]
```

you select the single character substring to the right of position i. Integer i may be positive or negative as long as its absolute value is no greater than the length

of the string. If i is out of range, subscripting fails. For example

`s [ 1 ]`	selects the first character in the string (if it has at least one character)
`s [ -1 ]`	selects the last character in the string (if it has at least one character)
`s [ 0 ]`	fails, there is no character following position 0.

You can assign to a substring of a variable. You cannot assign to a substring of a value: s[i]:="x" is okay; "abc"[i]:="x" is not.

6.4 Sectioning: subscripting ranges

If you subscript a string with a range

```
s [ i : j ]
```

you select the substring from position i to position j.

- Integers `i` and `j` may be positive or negative or zero in the range—length of the string up to length of the string plus one.
- The substring selected goes from the leftmost position to the rightmost position specified. Either `i` or `j` may specify the leftmost position. For example, `i` may be less than, equal to, or greater than `j`.
- If `i=j`, then an empty substring is selected.
- You can assign to a substring of a variable. You cannot assign to a substring of a value: `s[i:j]:="x"` is okay; `"abcde"[i:j]:="x"` is not.
- If `i` or `j` is out of range, subscripting fails.

For example

`s [ 2 : 0 ]`	selects all but the first character in the string (if it has at least one character)
`s [ -1:1 ]`	selects all but the last character in the string (if it has at least one character)

Another way to specify ranges is by specifying one position and the distance to the other position:

form	means
`s [ i +: j ]`	`s [ i : i + j ]`
`s [ i -: j ]`	`s [ i : i - j ]`

6.5 String operators

The binary string operators are concatenation and comparison.

- For the comparison operators, if one operand is a prefix of the other, the longer is greater than the shorter.
- If an operand is not a string, it is converted to one, if possible. If not possible, Icon causes a run-time error.

The unary operators for strings are length, generation, and random selection.

Table 12 String operators

operator	precedence	explanation
`s1 || s2`	7	concatenation
`s1 == s2`	6	equal
`s1 ~== s2`	6	not equal
`s1 << s2`	6	less than
`s1 <<= s2`	6	less than or equal
`s1 >> s2`	6	greater than
`s1 >>= s2`	6	greater than or equal
`* s`	12	length of the string
`! s`	12	generates the one-character substrings of s, equivalent to s[1 to *s]. If s is a variable, !s generates variables.
`? s`	12	produces a randomly generated one character substring of s. If s is a variable, ?s is a variable.

Example.

```
s:="abcde"
every !s:=""
write(s)
```

would write

```
bd
```

While

```
s:="abcde"
every !s:="xy"
```

would execute until it runs out of string space.

6.6 String editing and conversion functions

Icon excels in string handling. It provides a large collection of built-in procedures for string formatting and editing. In addition, several more are provided by the Icon Program Library (indicated by "link" in their specifications).

Table 13 String editing and conversion functions

`center(s,i)`	produces a string of length i containing string s centered in it with blanks appended to both sides to fill out the field. If *s>i, then it returns the middle i characters of s.
`center(s1,i,s2)`	produces a string of length i containing string s1 centered in it with copies of string s2 appended to both sides to fill out the field. If *s>i, then it returns the middle i characters of s.
`char(i)`	produces a one character string where the single character has the internal representation given by integer i, 0≤i≤255.
`compress(s,c)`	Let x be a character in set c. A substring of s composed entirely of character x is replaced with a single character x. (Character sets are presented in 6.8 on page 68.) link strings
`detab(s,i1,i2,...,in)`	copies string s replacing tab characters with blanks. The integer parameters give the tab stops. If more tab stops are needed, the last interval is repeated.
`entab(s,i1,i2,...,in)`	copies string s inserting tabs where possible. The integer parameters give the tab stops.
`image(s)`	produces a legible image of string s contained in double quotes. Characters \ and " are represented \\ and \". Special characters are represented in a form given in Table 11 on page 61, but if there is no \c representation available, then the \x*hh* form is used.

Table 13 String editing and conversion functions

`image(cs)`	produces a legible image of cset cs contained in single quotes. Characters \ and ' are represented \\ and \'. Special characters are represented in a form given in Table 11 on page 61, but if there is no \c representation available, then the \x*hh* form is used. (Character sets are presented in 6.8 on page 68.)
`image(n)`	produces the string representation of number n.
`image(x)`	produces a legible image of object x. For the mutable objects, the general format is "*type_num*(*size*)", where *type* identifies the type of object, *num* identifies the particular instance of that type, and *size* gives the number of elements it contains.
`left(s,i)`	produces a string of length i containing string s left justified with blanks appended to the right to fill out the field. If *s>i, then it returns s[1:i+1]
`left(s1,i,s2)`	produces a string of length i containing string s1 left justified with copies of string s2 appended to the right to fill out the field. If *s>i, then it returns s[1:i+1]
`map(s1,s2,s3)`	creates a new string which is a copy of s1 except for replacements made as follows: It replaces each character s1[i] that occurs in s2 at s2[j] with the character s3[j]. Strings s2 and s3 must be the same length. If the same character occurs more than once in s2, the rightmost occurrence determines the replacement character.
`mapstrs(s,l1,l2)`	replaces substrings. Lists l1 and l2 contain strings. Each occurrence of a string of l1 in s is replaced. An occurrence of the ith string of l1 in s is replaced by the ith string in l2. If l2 is shorter than l1, the rightmost, unpaired strings in l1 are deleted. In cases of overlap, the leftmost match is preferred. If two strings match at the same location, the longer is preferred. link mapstrs

Table 13 String editing and conversion functions

`repl(s,i)`	produces a string equal to i copies of s concatenated together
`replace(s1,s2,s3)`	replaces all occurrences of substring s2 in s1 by s3. link strings
`reverse(s)`	produces the string s reversed
`right(s,i)`	produces a string of length i containing string s1 right justified with blanks appended to the left to fill out the field. If *s>i, then it returns s[-i:0]
`right(s1,i,s2)`	produces a string of length i containing string s1 right justified with copies of string s2 appended to the left to fill out the field. If *s>i, then it returns s[-i:0]
`string(x)`	converts a number or a cset to a string.
`trim(s)`	produces a copy of string s with trailing blanks removed
`trim(s,cs)`	produces a copy of string s with all the rightmost characters that are contained in cset cs removed. (Character sets are presented in 6.8 on page 68.)
`type(x)`	produces a string naming the type of object s, one of: `"integer" "real" "string" "cset" "list" "table" "set" "procedure" "co-expression"` "window" or the name of a record type.

6.7 Idiom: map

The map function

```
map(s1,s2,s3)
```

copies string `s1`, replacing the characters that occur in `s2` with the characters at the same positions in `s3`. For example, `s:=map(s,"\t"," ")` will replace tabs in `s1` with blanks.

A different, and often more useful way to think of this is that the characters in s3 are placed into the form given in string s1. Characters of s3 may be moved

around, omitted, or have other characters inserted. For example, the keyword `&date` gives the current date in the form "*yyyy/mm/dd*". We can put it into the form "*mm/dd/yy*" as shown next. In the string `"abcdefghij"` characters *abcd* select the digits in the year, *fg* the month, and *ij* the day. Characters *e* and *h* select the slashes. The code

```
write(&date)
s:=map("fg/ij/cd","abcdefghij",&date)
write(s)
```

wrote out

```
1996/02/03
02/03/96
```

6.8 Character sets: cset

Character sets are used with string scanning procedures. You will often want to scan over a string of characters in a set, or up to any character in a set. You specify those character sets with the Icon *cset* type.

6.8.1 Character set literals

You write a character set literal surrounded by single quotation marks. Table 11 on page 61 shows the way to represent special characters in a cset literal—the form is the same as for string literals..

6.8.2 Character-set valued keywords

Several keywords have cset values.

Table 14 Keywords with cset values

`&ascii`	produces the character set containing all ASCII characters (128 characters).
`&cset`	produces the character set with all characters present (256 characters).
`&digits`	`'0123456789'`
`&lcase`	`'abcdefghijklmnopqrstuvwxyz'`
`&letters`	`'ABCDEFGHIJKLMNOPQRSTUVWXYZab_ cdefghijklmnopqrstuvwxyz'`
`&ucase`	`'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`

6.8.3 Character set operators

The cset operators are what you would expect for sets: Union, intersection,

complement, and relative complement, plus the size operator (the unary *).

Table 15 Character set operators

operator	precedence	explanation
`~ c`	12	a cset with the characters not in `c`
`* c`	12	the number of characters in the cset `c`
`c1 ** c2`	9	the intersection of character sets `c1` and `c2`, i.e., containing only those characters in both `c1` and `c2`
`c1 ++ c2`	8	the union of character sets `c1` and `c2`, i.e., containing those characters in either `c1` or `c2`
`c1 -- c2`	8	the difference of character sets `c1` and `c2`, i.e., containing only those characters in `c1` that are not in `c2`

6.9 String scanning functions

String scanning functions are used to search for patterns in strings. They generally behave as follows:

- The string scanning functions in Icon have the general form

```
function(x,s,i,j)
```

where the parameters represent

`x` what to look for,

`s` string to look in,

`i` starting position,

`j` ending position.

- They succeed if they find what they are looking for. They fail if they don't.
- Some of the functions—`any`, `bal`, `many`, `match`— expect to find what they are looking for at the starting position of the scan. If they succeed, they return the position just beyond the string they found.
- The other functions—`find` and `upto`— hunt through the string generating all the positions where they find what they are looking for.
- If the starting position i is to the right of position j, then the roles of `i` and `j` are reversed, i.e. the function behaves as if it were called

function(x,s,j,i).

Table 16 String scanning functions

`any(c,s)`	returns 2 if s[1] exists and is in character set c; otherwise it fails
`any(c,s,i)`	returns i+1 if s[i] exists and is in character set c; otherwise it fails
`any(c,s,i,j)`	returns i+1 if i<j and s[i] exists and s[i] is in character set c; otherwise it fails
`bal(c1,c2,c3,s)`	generates the positions k in s where 1≤k<*s+1 and s[k] (if it exists) is in cset c1, the number of characters in s[1:k] in cset c2 equals the number in c3, and there is no position m, 1≤m≤k, where the number of characters in s[1:m] in cset c2 is less than the number in c3.
`bal(c1,c2,c3,s,i)`	generates the positions k in s where i≤k<*s+1 and s[k] (if it exists) is in cset c1, the number of characters in s[i:k] in cset c2 equals the number in c3, and there is no position m, i≤m≤k, where the number of characters in s[i:m] in cset c2 is less than the number in c3.
`bal(c1,c2,c3,s,i,j)`	generates the positions k in s where i≤k<j and s[k] (if it exists) is in cset c1, the number of characters in s[i:k] in cset c2 equals the number in c3, and there is no position m, i≤m≤k, where the number of characters in s[i:m] in cset c2 is less than the number in c3.
`find(s1,s2)`	generates the positions k in s2 from 1 to *s2-*s1 which contain the beginning of the occurrences of s1, *i.e.*, where s2[k+:*s1]==s1. It fails if no occurrences of s1 are found.
`find(s1,s2,i)`	generates the positions k in s2 from i to *s2-*s1 which contain the beginning of the occurrences of s1, *i.e.*, where s2[k+:*s1]==s1. It fails if no occurrences of s1 are found.
`find(s1,s2,i,j)`	generates the positions k in s2 from i to j-*s1 at which s1 occurs as a substring, *i.e.*, where s2[k+:*s1]==s1. It fails if no occurrences of s1 are found.

Table 16 String scanning functions

`many(c,s)`	returns the position in s following the longest initial substring in cset c. Returns *s+1 if all the characters are in c. Fails if the first character of s isn't in c. (This saves you from having to write something like: (upto(~c,s) | (any(c,s)&*s+1)) \1.)
`many(c,s,i)`	returns the position in s following the longest initial substring in cset c beginning at position i. Returns *s+1 if all the characters are in c. Fails if s[i] isn't in c.
`many(c,s,i,j)`	returns the position in s following the longest initial substring in cset c beginning at position i and not extending beyond position j. Returns j if all the characters are in c. Fails if s[i] is not in c or if s[i:j] would fail (*i.e.*, the range is not valid).
`match(s1,s2)`	returns *s1+1 if s2[1+:*s1] == s1; otherwise fails
`match(s1,s2,i)`	returns *s1+i if s2[i+:*s1] == s1; otherwise fails
`match(s1,s2,i,j)`	returns *s1+i if s2[i+:*s1] == s1; otherwise fails. Requires position j to be at least *s to the right of position i or it will fail.
`segment(s,c)`	generates a sequence of strings which are the longest substrings of s from left to right composed solely of characters that alternatively do or do not occur in c. `link segment`
`slashbal(` `c1,c2,c3,s,i,j)`	like bal, but does not count a character from c2 or c3 that is preceded by a backslash character when determining balance. `link slashbal`
`slshupto(c,s,i,j)`	like upto, but treats backslash as an incorporation character in s, preventing the position of following character from being generated. Parameters s, i, and j default as in the built-in functions, but requires i≤j. (Warning: slshupto is reputed to have bugs.) `link slshupto`

Table 16 String scanning functions

`upto(c,s)`	generates the positions in s from 1 to *s which contain characters in set c. Fails if no such character is found.
`upto(c,s,i)`	generates the positions in s from position i to *s which contain characters in set c. Fails if no such character is found.
`upto(c,s,i,j)`	generates the positions in s from position i to position j which contain characters in set c. Fails if no such character is found.

6.10 Automatic conversions

Icon will automatically do conversions among numbers, strings, and character sets. See Chapter 13 on page 127.

- If a number is used in a context that requires a string, it is automatically converted to its string representation.
- If a string is used in a context that requires a number, it must contain a representation of a number. It is converted to a number if possible, otherwise there is a run-time error.
- If a character set is used in a context that requires a string, it is automatically converted to a string. The string will have each character in the set occurring once, sorted by their numeric representations' order.
- If a string is used in a context that requires a character set, it is automatically converted to a character set containing all the characters that appear in the string.
- If a number is used in a context that requires a character set, it is first converted to a string and then the string is converted to a character set.

6.11 Examples of strings

6.11.1 Finding the rightmost occurrence

You can find the first occurrence of a substring using `find`. You can find the last occurrence by putting the `find` in an every expression. In the following, we find the position of the rightmost `"."` in a string:

Figure 17 Find last occurrence

```
#find last occurrence of "."
i:=0
every i:=find(".",s)
```

6.11.2 Squeezing whitespace

Suppose we wish to remove leading and trailing whitespace from a string and

replace internal whitespace with single blanks. For our purposes, whitespace includes blanks and tabs. Here is code that will do it. First we replace tabs with blanks; then trim blanks from the right. The third line removes initial blanks. The fourth line repeatedly finds pairs of blanks and then replaces the longest string of blanks it can find at that position with a single blank.

Figure 18 Squeezing whitespace

```
s:=map(s,"\t"," ")
s:=trim(s)
s[1:many(" ",s)] := ""
while s[i:=find("  ",s):many(" ",s,i)]:= " "
```

6.11.3 Converting two hex digits to a character

Here's some code to convert a string of two hexadecimal digits to a character. If the argument is longer than two digits, the first two digits are used. The code uses the radix notation (see section 7.1 on page 83) to let Icon do the conversion of the hex number to an integer.

Figure 19 Converting two hex digits to a character

```
#hex to char
procedure hexToChar(h)
return char(integer("16r"||(h[1:3]|h)))
end
```

6.11.4 Converting a character to two hex digits

Here is code to convert a character to a pair of hexadecimal digits:

Figure 20 Converting a character to two hex digits

```
#char to hex
procedure charToHex(c)
local n
static hex
initial hex:= "0123456789ABCDEF"
n:=ord(c)
return hex[n/16+1]||hex[n%16+1]
end
```

6.11.5 Removing backspaces

Here is code to copy its input into its output, removing backspaces and the character preceding them. A backspace at the beginning of a line is simply left there.

Figure 21 Removing backspaces

```
#remove backspaces
procedure main()
while s:=read() do {
     while i:=find("\b",s,2) & s[i-1+:2] := ""
     write(s)
}
```



```
end
```

6.11.6 Generating character set tests for C

One great use for Icon is writing little tools to help with programming in other languages. Here is a subroutine to write out a C procedure to test whether a character is in a character set. It is given the name for the test procedure, the character set, and the number of bits in an unsigned integer for the destination system.

Figure 22 Generating character set tests for C

```
#gen C char set tests
procedure csetTestInC(name,cs,bitsPerInt)
local a, i, j, m, n, numPerLine
numPerLine := 4
m:=(256+bitsPerInt-1)/bitsPerInt
a:=list(m,0)
every c:=!cs & n:=ord(c) & i:=n/bitsPerInt &
j:=n%bitsPerInt do
     a[i+1]:=ior(a[i+1],ishift(1,j))
write("unsigned int ",name,"(int c) {")
write(" static unsigned int a[]={")
j:= numPerLine
every i:=1 to m-1 do {
     writes(a[i],",")
     j-:=1
     if j=0 then {write(); j:= numPerLine }
}
write(a[-1],"};")
write("return ((a[c/",bitsPerInt,"]>>(c%",bitsPer-
Int,"))&1);\n}")
end
```

The call

```
csetTestInC("letter",&letters,16)
```

will write out

```
unsigned int letter(int c) {
 static unsigned int a[]={
0,0,0,0,
65534,2047,65534,2047,
0,0,0,0,
0,0,0,0};
return ((a[c/16]>>(c%16))&1);
}
```

6.11.7 Generate identifiers

Here is a procedure to generate all the identifiers in a string.

Figure 23 ident: Generating identifiers

```
#generate identifiers in string
procedure idents(s)
local i, j, initIdChars, idChars
initIdChars := &letters++'_'
idChars := initIdChars++&digits
i := 1
while j := upto(initIdChars,s,i) &
     not any(idChars,s[j-1]) &
     k := many(idChars,s,j) do {
          suspend s[j:k]
          i:=k
}
end
```

Notice that we need to check that the character preceding the identifier could not itself be part of an identifier. When you have to do such checks, you might wish to consult the file lastc.icn in the Icon Program Library which has a routine to do such checks. It also has a routine to generate positions in a string of a specified substring delimited by characters from a specified set.

6.11.8 Primes sieve

Here is a program to write out the primes up to 1000 using the Sieve of Eratosthenes. Compare it to Figure 26 on page 87 which shows the same algorithm using bit operations.

Figure 24 Primes sieve using strings

```
procedure main()
local p,i,j,n
n:=1000
p:=repl("1",n)
every i:=2 to sqrt(n) do
     if p[i] =="1" then
          every j:= i+i to n by i do p[j]:="0"
every i:=2 to n do if p[i]=="1" then write(i)
end
```

6.12 Scanning Strings

6.12.1 Scanning

The functions `any`, `many`, `match`, `find`, and `upto` as presented in section 6.9 on page 69 are inconvenient to use. They require you name the string over and over again, although you are probably looking through only a single string at a time. They also require you to use integer variables to keep track of your position in the string, although each match typically starts where the previous one left off.

Icon allows you to get around these annoyances. It allows you to announce at a beginning of an expression what string you will be scanning and it keeps track

of the position in that string for you.

You specify which string you will be scanning with the binary `?` operator.

```
s ? e
```

means that within expression `e` you will be scanning the string `s`. You almost always need to follow the question mark by an expression sequence in braces.

The way Icon remembers which string you are scanning is by assigning it to keyword `&subject`. It remembers where you are in the string in the keyword `&pos`.

When you use the scanning operation `s?e`, Icon

- evaluates `s`
- saves the previous values of `&subject` and `&pos`
- assigns the value of s to `&subject` and the number 1 to `&pos`
- evaluates `e`
- restores the old values of `&subject` and `&pos`
- yields the value of `e` as the value of the scanning expression

Of course, backtracking into `e` will reestablish the values of `&subject` and `&pos` to continue the scan. Backtracking out of `e` will reestablish the values of `&subject` and `&pos` outside of the scanning expression and will backtrack into `s` to generate more strings to scan.

6.12.2 Functions tab and move

Although you can assign to `&pos`, the typical way of changing it is through the functions `tab` and `move`. Procedure `tab` sets `&pos` to an absolute position in `&subject`; `move` assigns `&pos` a position relative to its current value. Both procedures return the substring that `&pos` moved past, i.e., between its initial and final positions.

Procedure `tab` is used with the string scanning functions as described in the next section.

6.12.3 String scanning functions revisited

The string search functions described above in section 6.9 on page 69 will scan `&subject` from position `&pos` up to the end of the string if you do not specify any other string or position. That is to say, if you do not specify some other string:

they will search string `&subject`

they will start their operation at `&pos`

they will search the entire rest of `&subject` to the right of `&pos`

if they are successful, they return a new position just following what they scanned past.

you use `tab` to move `&pos` past the string that was scanned.

Table 17 String scanning, revisited

`any(c)`	any(c,&subject,&pos,0)
`bal(c1,c2,c3)`	bal(c1,c2,c3,&subject,&pos,0)
`find(s1)`	find(s1,&subject,&pos,0)
`many(c)`	many(c,&subject,&pos,0)
`match(s1)`	match(s1,&subject,&pos,0)
`move(i)`	moves `&pos` to position `&pos+i` in `&subject` and returns the substring between the original position of `&pos` and its new position. The new position can be zero or negative, but `&pos` is kept as a positive number. The assignment to `&pos` is reversible: when `move` is resumed during backtracking, `&pos` will be set back to its original position before the move.
`slashbal( c1,c2,c3)`	`slashbal(c1,c2,c3,&subject,&pos,0)` link slashbal
`slshupto(c)`	`slshupto(c,&subject,&pos,0)` link slshupto
`tab(i)`	moves `&pos` to position `i` in `&subject` and returns the substring between the original position of `&pos` and its new position. Position `i` can be zero or negative, but `&pos` is kept as a positive number. The assignment to `&pos` is reversible: when resumed during backtracking, `&pos` will be set back to its original position before the `tab`.
`upto(c)`	upto(c,&subject,&pos,0)

6.12.4 Matching a string, = e

The unary = operator tests to see if its operand string occurs next in the `&subject` string and moves `&pos` past it if it does, i.e.,


```
= e
```

is equivalent to

```
tab(match(e))
```

6.12.5 Scanning with assignment, ?:=

The scanning operator can be combined with assignment.

```
v ?:= e
```

initialized `&subject` and `&pos` from the string value of v. Expression `e` scans `&subject`. The value e produces is assigned to `v`.

You will often use this operator to replace a pattern in the middle of a string. Your code will probably look like this:

```
v ?:= tab over prefix ||
      (tab over pattern , replacement string) ||
      tab(0)
```

6.12.6 Testing &pos, pos(i)

The procedure `pos(i)` succeeds if `&pos` is at position `i` in `&subject`. Although you could test `&pos=i`, that will only work for positive values of `i`. Procedure pos also allows zero and negative positions to be specified, so the most common use is `pos(0)` to see if the entire string has been scanned.

6.12.7 Example

Here's a version of the `idents` procedure using string scanning operations:

Figure 25 idents with ? and tab

```
#generate identifiers in string
procedure idents(s)
local i, j, initIdChars, idChars
initIdChars := &letters++'_'
idChars := initIdChars++&digits
s ? suspend tab(upto(initIdChars)) &
          pos(1) | (move(-1),tab(any(~idChars))) &
      tab(many(idChars))
end
```

6.13 Regular expressions

Regular expressions are a way to express lexical patterns, e.g. numbers, identifiers, operators, and punctuation. UNIX software such as LEX and EGREP use a form of regular expressions for pattern matching.

The Icon Program Library provides two modules that deal with regular expressions: findre.icn and regexp.icn.

6.13.1 findre

File *findre.icn* defines one procedure,

```
findre(re,s,i,j)
```

where re is a string containing the regular expression, and s, i, and j are as usual. To use it, be sure to include `link findre`. Most characters in the regular expression represent themselves, but some have special meanings:

Table 18 Regular expression special characters

`.`	matches any single character.
`+`	matches one or more occurrences of the preceding element, e.g. `a+` matches one or more `a`'s.
`*`	matches zero or more occurrences of the preceding element, e.g. `a*` matches zero or more `a`'s. Procedure findre uses a shortest match first algorithm, so `*` will first match zero occurrences of the preceding pattern.
`()`	Groups the regular expression within the parentheses into a single element, usually for a following `*` or `+`.
`\`	incorporates the following character, removing its special meaning, e.g. `\+` matches a single `+` character; `\++` matches one or more `+`'s.
`|`	separates alternatives, e.g. `(0|1)+` matches a nonempty string of zeros and ones.
`[]`	Matches any one of the characters listed between the brackets, e.g. `[0123456789]` matches any digit.
`^`	At the beginning of a regular expression, `^` forces the expression to match at the beginning of the string. At the beginning of a character set, i.e. just following the `[`, it causes the set to be complement of the following characters, e.g. `[^0123456789]` matches any character except a digit.
`$`	At the end of a regular expression, `$` matches only at the end of the line.

As with find, findre is a generator, generating the leftmost positions at which the regular expression matches. The position to the right of the matching expression is assigned to global variable `__endpoint`, to which you can tab to get past the expression.

6.13.2 regexpr

File regexpr.icn contains three procedures of significance, shown in Table 19.

Table 19 Procedures in regexpr.icn.

`ReMatch(re,s,i1,i2)`	generates the positions in `s` following occurrences of regular expression `re` beginning at `i1`. Regular expression `re` can be a string representation of a regular expression, or a list representation created by procedure `RePat(s)`. `link regexp`
`ReFind(re,s,i1,i2)`	generates the positions in `s` of occurrences of regular expression `re`. The positions generated will be the leftmost positions of the matching strings. Regular expression `re` can be a string representation of a regular expression, or a list representation created by procedure `RePat(s)`. `link regexp`
`RePat(s)`	translates a string representation of a regular expression into a list representation. If you are going to use the same expression repeatedly, it is best to translate it with `RePat` once rather than having `ReMatch` or `ReFind` translate the string representation repeatedly. `link regexp`

The components of a regular expression used by module *regexp* are a superset of those provided by *findre*.

Table 20 Regexp special characters.

`.`	matches any single character.
`+`	matches one or more occurrences of the preceding element, e.g. `a+` matches one or more `a`'s.
`*`	matches zero or more occurrences of the preceding element, e.g. `a*` matches zero or more `a`'s. Module regexp by default uses a leftmost longest match first algorithm, so `*` will first match as many occurrences of the preceding pattern as it can.
`?`	matches zero or one occurrence of the preceding element.

Table 20 Regexp special characters.

`()`	Groups the regular expression within the parentheses into a single element, usually for a following `*` or `+`.
`\`	incorporates the following character, removing its special meaning, e.g. `\+` matches a single `+` character; `\++` matches one or more `+`'s.
`|`	separates alternatives, e.g. `(0|1)+` matches a nonempty string of zeros and ones.
`[]`	Matches any one of the characters listed between the brackets, e.g. `[0123456789]` matches any digit.
`-`	between two characters within a character set, `-` indicates all the ASCII character sequence, e.g. `[0-9]` represents any digit, `[A-Za-z0-9_]` represents any character that can occur in an Icon identifier.
`^`	At the beginning of a regular expression, `^` forces the expression to match at the beginning of the string. At the beginning of a character set, i.e. just following the `[`, it causes the set to be complement of the following characters, e.g. `[^0123456789]` matches any character except a digit.
`$`	At the end of a regular expression, `$` matches only at the end of the line.
`{N}`	where `N` is a number matches exactly N occurrences of the preceding element.
`{N,}`	where `N` is a number matches N or more occurrences of the preceding element.
`{M,N}`	where `M` and `N` are numbers, matches no fewer than M and no more than N occurrences of the preceding element.
`\N`	where `N` is a single digit from one to nine, matches the same string that a preceding parenthesized subexpression matched. The parenthesized expression selected is the one beginning with the Nth "`(`" counting from the left.
`\w`	matches any character that can occur in an Icon identifier, i.e. `&letters ++ &digits ++ '_'`.
`\W`	matches any character not matched by `\w`.
`\b`	matches at a word boundary, i.e. between a character in `\w` and `\W`, in either order.
`\B`	matches anywhere `\b` does not, i.e. within strings of `\w` or `\W` characters.

Table 20 Regexp special characters.

`\s`	matches any whitespace character.
`\S`	matches any character that isn't whitespace.
`\d`	matches any digit, i.e. is equivalent to `[0-9]`.
`\D`	matches any non-digit, i.e. is equivalent to `[^0-9]`.

Chapter 7 Arithmetic

Like most languages, Icon provides both integer and real data types. Unlike most languages, Icon provides integers of unbounded precision.

7.1 Numeric literals

Integer literals (*constants*) can be written in either decimal or radix format. A decimal literal is written just as a string of digits, *e.g.*,

```
1066
0
025
```

A radix literal can be used to write an octal constant, or hexadecimal, or any radix from 2 to 36. It is written as the radix, in decimal, followed by the letter R (in either upper or lower case) followed by a string of digits and letters that specifies its digits. The letter A represents 10; B, 11; all the way to Z, 35. Upper and lower case letters are considered the same. For example,

```
8r31
16R19
30rP
2r11001
```

Notice that there are no negative integer literals. If you write something like **`-10`**, you are really applying the `-` operator to the literal **10**.

Real literals are written in decimal using either decimal point or an exponent or both, e.g.,

```
25.0
25e0
2.5E+1
250e-1
```

Real literals are generally of the form given by the following grammar:

real_literal = digits “.” [digits] [(“e”|”E”)[“-”|”+”] digits.

digits = digit {digits}.

digit = “0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”.

Note: The grammar is **not** part of Icon; it is used to describe Icon. In the grammar, all literal characters are quoted. The equal sign defines the name on its left hand side to match the pattern on its right. The vertical bar separates alternatives. Parentheses' group alternatives. Brackets enclose things that may or may not be present. Braces enclose things that may be present any number of times or may be absent entirely.

In a recent version of Icon, real literals were allowed to begin with a decimal point rather than a digit. In previous versions, they were not. Now one-half can be written `.5` rather than `0.5`.

7.2 Operators

Icon provides the normal arithmetic operators.

Of the binary operators, exponentiation (**^**) is performed before multiplication, division, and remainder (*****, **/**, **%**), which are performed before addition and subtraction (**+** and **-**).

A binary operation is performed in real arithmetic if either operand is real. If both operands are integer, it is performed as an integer operation.

Exponentiation associates to the right while the other binary operators associate to the left. It makes more sense that way: 3^3^3 = 3^(3^3) = 3^27 =7625597484987. If it associated to the left, it would yield (3^3)^3 = 3^(3*3) = 3^9 = 19683.

You can use a string as an operand: it will automatically be converted to the number it represents. If the string does not represent a number, the program will stop and write out an error message. (Icon can convert negative numbers.)

There is a random number generator operator, **?**. When applied to a positive integer n, ?n produces a randomly chosen integer in the range 1 to n. When applied to zero, it produces a random real in the range 0.0 to 1.0.

Table 21 Arithmetic operators

operator	precedence	explanation
`+ x`	12	numeric value. If x is a number, it is left unaltered. If it is the string representation of a number, it is converted to the corresponding number.
`- x`	12	negative
`? i`	12	produces a random integer in the range 1 to i if i is an integer greater than zero. produces a random real number in the range 0.0 to 1.0 if i = 0.
`e1 ^ e2`	10	exponentiation. **Right associative.**

Table 21 Arithmetic operators

operator	precedence	explanation
`e1 * e2`	9	multiplication
`e1 / e2`	9	division
`e1 % e2`	9	remainder. The sign of result is the sign of e1. This operation works for both integer and real. For real operands, e1%e2=e1-integer(e1/e2)*e2.
`e1 + e2`	8	addition
`e1 - e2`	8	subtraction

All binary operators (except assignment itself) can be combined with the assignment operator in the form

op`:=`

to perform the operation on the left and right operands and assign the result to the left. The most common use, no doubt, is

```
i +:= 1
```

which increments variable i.

7.3 Large integers

Icon allows integers to be arbitrarily large, but it uses a more efficient representation for standard sized integers up to the wordsize of the computer. There are a few problems with large integers:

- *to* and *seq* do not work with large integers.
- large integer literals are converted from character strings when they are encountered in the program. You should avoid writing them in loops.
- Converting large integers to character strings can take a long time.
- Not all Icon implementations provide large integers.

If you want to test whether an integer, `i`, is a large integer, do the following:

- include in your program

```
link large
```

- call

```
large(i)
```


which returns the value of `i` if `i` is a large integer and fails if `i` is not.

7.4 Conversion functions

There are four built in functions that convert values to an integer or real:

Table 22 Built-in number conversion

`numeric(x)`	will convert a string representation of a number to the numeric representation. It will leave a number unaltered. It will fail if the conversion is not possible.
`integer(x)`	will convert a real to an integer, or a string representation of a number to an integer. Even if x is a string representation of a real, it will be converted to an integer. An integer is left unaltered. It will fail if the conversion is not possible.
`real(x)`	converts to a real, but is like integer(x) otherwise. It will fail if the conversion is not possible.
`ord(s)`	takes a one character long string and converts the character to the integer that represents it in the character set. For example, in ASCII, `ord("A")` = `65`.

The integer, real, and numeric functions will *fail* if their operands cannot be converted. The program can catch the failure and take some appropriate action. If you just pass the operand to an arithmetic operator and it cannot be converted, the program will terminate with an error message.

There are several more conversion functions available in the Icon Program Library to convert reals to integers. To use them, include the linkage declaration

```
link real2int
```

Table 23 Other conversions from real to integer

`ceil(r)`	nearest integer to r away from 0
`floor(r)`	nearest integer to r toward 0
`round(r)`	nearest integer to r
`sign(r)`	sign of r: -1 if r is negative, 0 if r is 0, 1 if r is positive.
`trunc(r)`	nearest integer less than r

7.5 Bitwise operations on integers

Integers are represented as bit strings. If you number the bits from 0 at the right, bit number i contributes 2^i to the value of the integer. In twos-complement representation, the leftmost bit in the n-bit number contributes not 2^{n-1} but its negative, -2^{n-1}. Icon provides a collection of functions to perform the usual bit-by-

bit operations on integers. In other languages these operations would be used to represent sets, but Icon provides a more convenient set data type, see Chapter 11 on page 117.

Table 24 Bitwise operators

`iand(i,j)`	bitwise **and**: a bit is set in the integer result only if it is set in both i and j.
`icom(i)`	bitwise **complement**: a bit is set in the integer result if and only if it is *not* set in i.
`ior(i,j)`	bitwise **or**: a bit is set in the integer result if it is set in either i or j.
`ishift(i,j)`	shift the bits in i by j positions to the left (if j>0) or |j| to the right (j<0), filling with zeros.
`ixor(i,j)`	bitwise **exclusive or**: a bit is set in the integer result only if it is set in one or the other but not both of i and j.

Example. Here is a program to compute primes by the "sieve of Eratosthenes." We set the bits in a long integer to represent their bit positions being potential primes. (That is, we use the integer as a bit set.) Starting at 2, we examine bits. If bit, i, is set (tested by `iand(p,ishift(1,i))~=0`), it represents a prime. We go through all multiples of that prime (`j:= i+i to n by i`) clearing those bits, since those bits obviously represent composite numbers.

Figure 26 Prime sieve using bits

```
procedure main()
local p,i,j,n
n:=1000
p:=icom(0)

every i:=2 to sqrt(n) &
      iand(p,ishift(1,i))~=0 &
           j:= i+i to n by i do
               p:=iand(p,icom(ishift(1,j)))
every i:=2 to n do if iand(p,ishift(1,i))~=0 then
write(i)
end
```

7.6 Numeric functions

Icon provides the common trigonometric and log functions. Note that Icon expresses angles in radians, rather than degrees. It provides conversion functions, **dtor** and **rtod**, to convert degrees to radians or vice versa. Hyperbolic functions

are available in the IPL file hyperbol.icn.

Table 25 Trig. and numeric functions and keywords

`abs(r)`	absolute value
`acos(r)`	arc cosine in radians, $-1 \le r \le 1$.
`asin(r)`	arc sine in radians, $-1 \le r \le 1$.
`atan(r1,r2)`	arc tangent of r1/r2 in radians with the sign of r1.
`atan(r)`	arc tangent of r in radians.
`cos(r)`	cosine of r (given in radians)
`cosh(r)`	hyperbolic cosine. `link hyperbol`
`dtor(r)`	degrees to radians
`&e`	The base of the natural logarithms. Approximately 2.71828182845904
`exp(r)`	e^r, or in Icon, &e^(r)
`log(r1,r2)`	logarithm of r1 to the base r2
`log(r)`	$\log_e r$
`&phi`	phi, the "golden ratio." Approximately 1.61803 = a/b where a/b=(a+b)/a
`&pi`	π, approximately 3.14159265358979
`&random`	The seed of the random sequence. You can assign a new value to it.
`rtod(r)`	convert radians to degrees
`sin(r)`	sine of r (given in radians)
`sinh(r)`	hyperbolic sine. `link hyperbol`
`sqrt(r)`	square root of real $r \ge 0$.
`tan(r)`	tangent of r (given in radians)
`tanh(r)`	hyperbolic tangent. `link hyperbol`

7.7 Complex

Complex numbers are not built in to Icon, but are provided in the IPL file `complex.icn`, which is to say, if you want to use them, include

```
link complex
```

in your program. The complex numbers are represented internally as records of type

```
record complex(rpart,ipart)
```

The string representation of complex numbers is given by the grammar:

["+" | "-"] number ("+" | "-") number "i"

which is to say, a complex number is an optional plus or minus sign, followed by a number, followed by a plus or minus sign, followed by another number, followed by the letter "i".

The procedures to perform operations on complex numbers are as follows:

Table 26 Complex arithmetic procedures.

`complex(r,i)`	create complex number with real part r and imaginary part i
`cpxadd(x1,x2)`	add complex numbers x1 and x2
`cpxdiv(x1,x2)`	divide complex number x1 by complex number x2
`cpxmul(x1,x2)`	multiply complex number x1 by complex number x2
`cpxsub(x1,x2)`	subtract complex number x2 from complex number x1
`cpxstr(x)`	convert complex number x to string representation
`strcpx(s)`	convert string representation s of a complex number to it’s internal representation

7.8 Rational numbers

The Icon Program Library contains a package to manipulate rational numbers, i.e. numbers that are expressed as the ratio of two integers. Internally, the rational numbers are represented as records:

```
record rational(numer,denom,sign)
```

To use rational numbers, you will need to include the linkage declaration:

```
link rational
```


The available procedures are:

Table 27 Rational arithmetic procedures.

`str2rat(s)`	Convert the string representation of a rational number (such as "3/2") to a rational number.
`rat2str(r)`	Convert the rational number r to its string representation.
`addrat(r1,r2)`	Add rational numbers: r1+r2.
`subrat(r1,r2)`	Subtract rational numbers: r1 - r2.
`mpyrat(r1,r2)`	Multiply rational numbers: r1 * r2.
`divrat(r1,r2)`	Divide rational numbers: r1 / r2.
`negrat(r)`	Negate a rational number: -r.
`reciprat(r)`	Get the reciprocal of rational number: 1/r.

The rational number package itself links to gcd.icn for a greatest common divisor routine. Routines find the greatest common divisor and the least common multiple are available in `gcdlcm.icn`.

7.9 Random numbers

Icon has a built-in random number generator accessed by the unary question mark operator.

Table 28 The random number generator, ? n.

`? 0`	yields a random real number in the range $0.0 \le ?0 < 1.0$.
`? n`	yields a random integer in the range $1 \le ?n \le n$, for integer $n>0$.
`? x`	yields a randomly chosen element from a structure, e.g. string or list.

The ? operator is not a generator. It will not generate another random number when backed into. For that you can use `|?n` or some procedures in the IPL.

The "seed" for the random number generator is the value of the keyword &random. Keyword &random will change after each application of the ? operator. It is the value of &random in its range (zero through some large value) that is converted into a random value returned by ?. Keyword &random starts at zero each program execution, but it can be assigned other values to avoid having all executions use the same sequence of pseudo-random values.

There are a number of files in the IPL that relate to random number generation.

Table 29 Random number packages in the IPL.

procedure	*description*
`gauss()`	returns a random number chosen from a gaussian distribution with a mean of zero `link gauss`
`gauss_random(x,f)`	returns a random number chosen from a gaussian distribution with a mean of x. Larger values of parameter f will flatten the distribution. `link gauss`
`randomize()`	a procedure to set the seed of the random number generator to a value determined in part from the date and time. You can use this to avoid always generating the same sequence of random numbers each time the program is run. `link randomiz`
`randreal(low,high)`	returns a random real number, r, in the range low ≤ r < high. `link randreal`
`ranseq(seed)`	*generates* the values of &random starting at seed. `link randseq`
`ranrange(min, max)`	returns a random integer in the range min to max, inclusive. `link ranrange`

You may wish to consult the following files in the IPL as well:

- File `random.icn` contains Icon code to perform the same functions as the built-in random number generator as well as to use different parameters.
- File `lcseval.icn` contains a procedure to evaluate parameters for congruential random number generators.

7.10 Matrices

Icon does not have multi-dimensional arrays or matrices built in. They are built using lists, see Chapter 9 on page 105. You can use the syntax A[i,j] instead of A[i][j] to subscript two levels of lists, which gives the feel of matrices.

The Icon Program Library does provide some matrix and linear algebra procedures. See `matrix.icn` and `lu.icn` in the Icon Program Library.

Chapter 8 I/O

8.1 File I/O

Values of type file represent open files on which the program can read and write. The program starts execution with three open files: `&input`, `&output`, and `&errout`. Unless another file is specified, read functions read from `&input`, write functions write to `&output`, and error messages go to `&errout`.

To open a new file for input or output, you call the procedure open which will return a file object that you pass to read or write to tell it which file to access, *e.g.*,

```
f:=open("x.txt","r")|
    stop("cannot open x.txt")
```

The open will fail if the file can't be opened for reading. This code tests for failure and terminates execution with a message if it fails.

File names follow conventions of the operating system Icon is running on. However, Version 9 for MSDOS allows UNIX path specifications, *i.e.*, using "/" rather than "\".

File I/O in Icon is based on UNIX. In UNIX a file may be opened for reading or writing. When opened for writing, the file position may be set at the beginning, which would replace the contents, or at the end for appending. The open may specify that the file be created, which requires the file not already exist. (Unless you're the superuser, but that's another matter.)

In UNIX, lines of text files are terminated by a newline character. When Icon reads a line (function `read`) it strips the newline character off the end and returns the line as a string. Function `reads`, however, will return the newline character like any other. When Icon is running on some other system, the open function by default specifies that that system's newline conventions be translated into UNIX's, for example, translating carriage return/line feed sequences into new lines. To read binary files, you must specify the `"u"` option to tell `open` not to mess with the actual bytes, e.g.,

```
open(filename,"ru")
```

In addition to the built-in functions and procedures, the Icon Program Library provides several useful procedures for file I/O. They will be indicated, as al-

ways, with a link command that needs to be used to access them.

Table 30 Operations and functions on files

`! f`	generates the lines of file f. Fails on end of file.
`close(f)`	closes the file bound to file object f.
`display(i,f)`	writes to file f the names of the i most recently called active procedures, their local variables, and the global variables. Used for debugging.
`display(i)`	writes to `&errout` the names of the i most recently called active procedures, their local variables, and the global variables. Used for debugging.
`display()`	writes to `&errout` the names of all the active procedures, their local variables, and the global variables. Used for debugging.
`dopen(s)`	opens the file named `s` using default options (i.e. `open(s,"rt")`). If the file is not found in the current directory, all the directories whose paths are listed in environment variable DPATH are tried, left to right until the file can be successfully opened. The paths in `DPATH` are separated from each other with blanks; the directories within the paths are separated by "`/`" characters. `link dopen`
`&errout`	the standard error output file. (It is not a variable; it cannot be reassigned.)
`flush(f)`	Output is typically buffered before being written. `flush(f)` flushes (actually writes out) the buffers for file `f`.
`&input`	the standard input file. (It is not a variable; it cannot be reassigned.)

Table 30 Operations and functions on files

`open(s1,s2)`	opens the file named by string `s1` for access in the mode described by string `s2` and returns a file object that represented it, or fails if it cannot be opened. The modes are indicated by letters: • `"a"`—open in append mode for writing • `"b"`—open for both reading and writing • `"c"`—create • `"r"`—open for reading (default) • `"w"`—open for writing • `"t"`—translate line terminations into linefeed characters (default) • `"u"`—do not translate line terminations to linefeed characters (use this for binary files) • "p"—create a process to execute command line s1 and attach it as a pipe to the current process. With "pr", the current process can read the standard output of the created process; with "pw", the lines the current process writes to the file can be read by the created process as its standard input.
`open(s,"pr")`	forks a process which executes the command line contained in string `s` and returns a file (bound to a pipe) from which the output of the forked process may be read. If your system doesn't have pipes, use `popen.icn` in the IPL.
`open(s,"pw")`	forks a process which executes the command line contained in string `s` and returns a file. Lines written to the file are piped as standard to the forked process. If your system doesn't have pipes, use `popen.icn` in the IPL.
`open(s1)`	is equivalent to `open(s1,"rt")`

Table 30 Operations and functions on files

`&output`	the standard output file. (It is not a variable; it cannot be reassigned.)
`pclose(file)`	closes the pipe bound to `file`, which was opened by `popen()`. `link popen`
`popen(s1,s2)`	equivalent to `open(s1,"p"||s2)` on systems with pipes. On systems without pipes, it will use the `system()` function and a temporary file to simulate a pipe. However, the command given in s1 will *not* run concurrently with the current process. (If you use `popen(s1, "w")`, you must use `pclose(file)` to actually have the command `s1` execute.) `link popen`
`read()`	reads and returns as a string the next line from the standard input file (`&input`), but fails on end of file. `read` strips off the terminating newline character from the line it returns.
`read(f)`	reads and returns as a string the next line from the file, `f`, but fails on end of file. `read` strips off the terminating newline character from the line it returns.
`reads()`	reads and returns as a string the next character from the standard input file (`&input`), but fails on end of file.
`reads(f)`	reads and returns as a string the next character from the file, `f`, but fails on end of file.
`reads(f,i)`	reads and returns as a string the next i characters from the file, `f`. Fails on end of file. Returns fewer than i characters if only that many remain.
`save(s)`	saves the currently executing program as file s and returns the size of the file created. When executed, the program will resume executing by returning from the `save`. *Not available on all systems.*

Table 30 Operations and functions on files

`seek(f,i)`	seeks to position `i` in file `f` so that subsequent reads or writes will start at the i-th byte. Fails if the seek cannot be done. As in Icon strings, the first byte in the file is at position 1, and the last byte is indicated by position 0.
`stop(x1,x2,...)`	writes out the values `x1,x2,...` left-to-right to the error output, `&errout`, and exits with an error status. If any `xi` is a file, subsequent output is to that file.
`where(f)`	returns the current file position, most likely for use with `seek` later.
`write(x1,x2,...,xn)`	writes out the values `x1,x2,...` left-to-right to the standard output, and follows them with a line termination. If any `xi` is a file, the following values are written to that file until the file is changed again or the end of the write procedure. If any `xi` is neither a file nor a string and cannot be converted to a string, `write` terminates program execution with an error. Returns xn.
`writes(x1,x2,...,xn)`	writes out the values `x1,x2,...` left-to-right to the standard output. It does *not* follow them with a line termination. If any `xi` is a file, the following values are written to that file until the file is changed again or the end of the write procedure. If any `xi` is neither a file nor a string and cannot be converted to a string, `writes` terminates program execution with an error. Returns xn.

Table 30 Operations and functions on files

`xdecode(f)` `xdecode(f,p)`	reads, reconstructs, and returns the Icon data structure from file `f` that was previously saved there by xencode. Files, co-expressions, and windows are decoded as empty lists (except for files `&input`, `&output`, and `&errout`). Fails if the file is not in xcode format or if it contains an undeclared record. If p is provided, xdecode reads the lines by calling `p(f)` rather than `read(f)`. See `xencode` for an idea of what to use this for. `link xcode`
`xdecoden(x,fn)`	like `xdecode`, except that `fn` is the name of a file to be opened for input (with `open(fn)`). `link xcode`

`xencode(x,f)`
`xencode(x,f,p)`
	encodes and writes the data structure `x` into file `f`. The data structure can be read back in by `xdecode`. If parameter p is provided, it is called in place of write, *i.e.* `p(f,...)` instead of `write(f,...)`, in which case `f` need not be a file, *e.g.*
	`xencode(x,L:=[],put)`
	will encode the data structure into a list, `L`.
	
	`link xcode`

`xencoden(x,fn,opt)`	like `xencode`, except that `fn` is the name of a file to be opened for output (with `open(fn, opt)`). The options, `opt`, default to `"w"`. `link xcode`

8.2 File names and paths

Full file names in most systems are called "paths" since they represent a path through the hierarchical directory system. *E.g.*

D:\IPL\PROCS\BASENAME.ICN

Moreover file names proper are usually divided into a base name and an extension (*e.g.* basename.icn). The Icon Program Library has several procedures to

break out the components of a file name.

Table 31 File names and paths: IPL procedures.

`basename(path, suffix)`	returns the base name of the file indicated by `path`. The `suffix` string is removed from the right. E.g.
	`basename("D:\IPL\PROCS\BALQ.ICN", ".ICN")`
	returns "BALQ". Works for UNIX, MSDOS, and MACs.
	`link basename`

`components(s,sep)`
`components(s)`
	returns a list of the components of the path `s`, where the components of the path are separated by the character `sep`. The separator defaults to "/" which is appropriate for UNIX. E.g.
	`components("/a/b/c.d")`
	returns
	`["/","a","b","c.d"]`
	`link filename`

`dpath(s)`	returns the path for the file whose file name is s. If the file is not found in the current directory, all the directories whose paths are listed in environment variable DPATH are tried, left to right until the file can be successfully opened. The paths in `DPATH` are separated from each other with blanks; the directories within the paths are separated by "`/`" characters. (Icon on MSDOS allows "`/`" rather than "`\`" in paths.) Procedure `dpath` returns • `s`, if the file is found in the current directory. • `path || "/" || s` , if the file is found at `path` within `DPATH`. `link dpath` or `link dopen` See also: `pathfind`.
`getpaths(p1, p2,...,pn)`	generates `p1`, `p2`, ..., `pn` followed by all the paths in the `PATH` environment variable. This will work for both UNIX and MSDOS, choosing the correct PATH syntax for each.

Table 31 File names and paths: IPL procedures.

`pathfind(s,p)`	returns the path for the file whose file name is `s`. If the file is not found in the current directory, all the directories whose paths are listed in string `p` are examined, left to right. If `p` is &null (*i.e.* not specified), the paths in environment variable DPATH are tried, left to right until the file can be successfully opened. The paths in `p` and `DPATH` are separated from each other with blanks; the directories within the paths are separated by "`/`" characters. (Icon on MSDOS allows "`/`" rather than "`\`" in paths.) Procedure `dpath` returns • `s`, if the file is found in the current directory. • `path || "/" || s` , if the file is found at `path` within `p` or `DPATH`. `link pathfind` See also: `dpath`.
`suffix(s,sep)` `suffix(s)`	returns the list `[pre,post]` where `pre` is the substring of `s` up to the last occurrence of `sep` and `post` is the substring of `s` to the right of the `sep`. The separator defaults to "`.`", appropriate for both UNIX and MSDOS. If the separator `sep` does not occur, `suffix` returns `[s,&null]`. `link filename`

Table 31 File names and paths: IPL procedures.

`tail(s,sep)`
 	
`tail(s)`
	returns the list `[pre,post]` where `pre` is the substring of `s` up to the last occurrence of `sep` and `post` is the substring of `s` to the right of the separator `sep`. The separator defaults to "`/`" which is appropriate for UNIX paths. Since Icon allows MSDOS paths to be specified with "/" rather than "\", it can be used for DOS if you translate the paths. There are a number of special cases, `tail` returns
	• `["",s]` if `sep` does not occur in `s`.
	• `[sep,s[2:0]]` if `sep==s[1]`.
	• `[s[1:j],s[j+1:0]]` if `sep` occurs at position `j`, `1<j<*s-1`.
	• `[s[1:-1],&null]` if `sep` occurs as the last character in `s`.
	
	`link filename`

`tempname()`	generates names for a temporary file, *i.e.* a file that does not appear to already exist. Under UNIX, the file name has the form
	`/tmp/icontmp.`*ddd*
	where *ddd* is a string of exactly three digits. Under MS-DOS, the filename is either of the forms:
	temp`\icon0`*ddd*`.tmp`
	or
	`icon0`*ddd*`.tmp`
	The first form uses the directory bound to the environment variable TEMP. If TEMP is not defined, then the second form is used, placing the file in the current directory.
	
	Because Icon cannot directly test whether a file exists, `tempname` returns the names of files it could not open for reading, which might mean the file exists but is locked. In that case, you will not be able to open it for writing either. Therefore `tempname` is a generator so that if you can not open the first file generated, you should be able to open a subsequent one.
	
	`link tempname`

8.3 Directories

There are several built-in Icon functions that manipulate the directory structure, changing the current directory or removing or renaming files. Several more are available in the Icon Program Library.

UNIX and DOS and probably most systems have environment variables. The system maintains a table of names bound to string values. The variables are used to keep information about the user's environment, such as terminal type and search paths for programs. Icon gives access to environment variables via the `getenv` function.

Table 32 Directory and environment procedures

`chdir(s)`	changes the current directory to that indicated by string s. Fails if it cannot change to that directory, perhaps because it does not exist.
`exists(name)`	succeeds if file named `name` can be opened, otherwise fails. `link exists`
`gdl(dir)`	returns a list of all the file names in the directory indicated by the string `dir`. Fails if there are no files in the directory. Works with UNIX and MSDOS. Includes the directory in the file names. `link gdl2`
`gdlrec(dir)`	(recursive gdl) returns a list of all the file names in the directory indicated by the string `dir` and all its sub directories. Fails if there are no files in the directory. Works with UNIX and MSDOS. Includes the directory in the file names. `link gdl2`
`getenv(s)`	Systems typically provide environment variables: a table mapping string names into string values. `getenv(s)` returns the string associated with environment variable s, or fails if there is none such.
`remove(s)`	removes the file named `s` from the disk directory, or fails if `s` cannot be removed.
`rename(s1,s2)`	renames the file whose name is `s1` to have name `s2`. Fails if it cannot rename `s1`.

8.4 Character-based, interactive I/O

On some systems (not all) the Icon program has direct access to the terminal. You can use these functions to write interactive systems. They do not work that

well on UNIX systems, however, and may leave the console in a strange mode if Icon terminates abnormally.

> **Warning:** These are not available in all versions of Icon. Moreover, windows (Chapter 17 on page 151) make these functions obsolete.

Table 33 Interactive character I/O functions

`getch()`	reads a character from the keyboard, but does not echo it. Waits until a character is available.
`getche()`	reads a character from the keyboard and echoes it. Waits until a character is available.
`kbhit()`	succeeds if a character has been typed at the keyboard that has not been read in yet. Use this to avoid waiting.

If you are trying to use ANSI terminals, consult the IPL modules: ansi.icn, iolib.icn, iscreen.icn, and itlib.icn.

Chapter 9 Lists

9.1 Creation: list(), [...]

You can create lists using either the function `list` or the bracket notation as shown in the following table.

Table 34 List creation

`[ ]`	create an empty list
`[e1,e2,...,en]`	create a list of n elements initialized to the values of the expressions `e1`,`e2`,...,`en`.
`list()`	create an empty list
`list(n)`	create a list of *n* elements all initialized to `&null`
`list(n,val)`	create a list of *n* elements all initialized to the value of `val`

Be warned that when you create a list using the form

```
list(n,val)
```

that `val` is evaluated once before the list is created. If `val` yields a mutable object, all list elements share it. For example,

```
L:=list(2,list(2))
L[1][1]:=1
```

will result in L[2][1] also equaling 1. If you actually want a two-dimensional array, use:

```
L:=list(2)
every !L:=list(2)
```

9.2 Positions subscripting and subranges

You can use lists as one-dimensional arrays.

The positions in a list are the same as in strings. The positions are at the left end, the right end, and between elements. If there are n elements in the list, the left

end is numbered 1 and -n, the right end is numbered n+1 and 0. The internal positions are numbered up from the left or down from the right. For example, [“F”, 22, 3.0, “g”] would be numbered as shown in section Figure 27 on page 106.

Figure 27 Character positions

[	“F”	,	22	,	3.0	,	“g”	]
1		2		3		4		5
-4		-3		-2		-1		0

A single subscript

```
L[i]
```

selects the element of the list to the immediate right of its position. The selected element is a variable, *i.e.,* it can be assigned to.

A subrange

```
L[i:j]
L[i+:j]
L[i-:j]
```

specification selects a sequence of elements. The selected elements are *not* a variable; a subrange *cannot* be assigned to. The selected subrange is *copied* as a new list.

If you are indexing into a series of lists (and/or tables or strings), you can use either

```
L[i][j][k]
```

or

```
L[i,j,k]
```

9.3 Operators

The three unary operators that apply to all structured types apply, of course, to lists—the size operator, *, the element generator, !, and the random selection, ?. List concatenation uses three vertical bars, |||, rather than string concatenation's two. The identity test operator, === (and its complement,~===), will succeed (or fail) if the two operands are the same object so that any changes to one will be seen through the other.

Table 35 List operators

`! L`	generates each element of the list as a variable

Table 35 List operators

`* L`	returns the length of the list
`? L`	returns a randomly selected element of the list as a variable
`L ||| M`	returns a new list equal to the concatenation of the two lists
`L === M`	succeeds if the two operands are the same list
`L ~=== M`	succeeds if the two operands are different lists

Example. Here's a clever way to randomize the order of elements in a list x:

```
every !x :=: ?x
```

although it does not randomize well. Ward Cunningham and Ralph E. Griswold, in procedure *shuffle* in the Icon Program Library, use

```
every i := *x to 2 by -1 do
    x[?i] :=: x[i]
```

Example. Here's how you can reverse a list, L, in place:

```
every i:=1 to *L/2 do L[i]:=:L[-i]
```

Example. Here's how you can rotate a list, L, by k places to the left:

```
L := L[k+1:0]|||L[1:k+1]
```

9.4 Stacks and queues

Lists may be used as doubly ended queues: you can insert or remove items from either end. You can use them as stacks or queues. The operations are pictured in Figure 28 on page 108 and listed in Table 36.

Table 36 Lists as doubly ended queues

`get(L)`	removes and returns the first element of list L
`pop(L)`	removes and returns the first element of list L
`pull(L)`	removes and returns the last element of list L

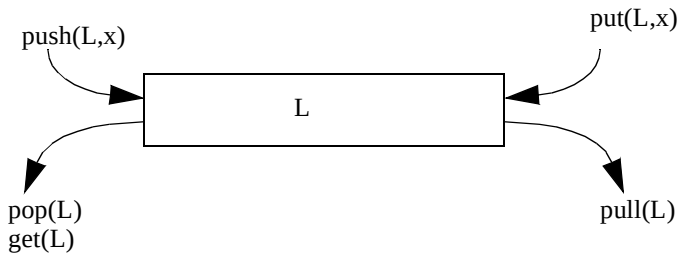


Figure 28 Stack and queue operations on a list.

Table 36 Lists as doubly ended queues

`push(L,x)`	inserts x as the new first element of list `L`, moving the other elements up one position, e.g., `push([1,2,3],4)` creates the same list as `[4,1,2,3]`.
`push(L,x1, x2, ..., xn)`	is equivalent to `{push(L,x1); push(L,x2); ... ; push(L,xn)}`. The end result is `xn` on top of the stack.
`put(L,x)`	inserts x as the new last element of list `L`, leaving the other elements in their previous positions, e.g., `put([1,2,3],4)` creates the same list as `[1,2,3,4]`.
`put(L, x1, x2, ..., xn)`	is equivalent to `{put(L,x1); put(L,x2); ... ; put(L,xn)}`.

Example. Here's how you can create a list, M, that is the reverse of another list, L:

```
M:=[ ]
every push(M,!L)
```

Example. Here is the primes sieve program using a list rather than a string or a long integer to keep track of the sets of candidate primes and known composite numbers. Compare to Figure 26 on page 87 and Figure 24 on page 75.

```
procedure main()
local p,i,j,n
n:=1000
p:=list(n,1)
every i:=2 to sqrt(n) do
```



```
        if p[i]=1 then
             every j:= i+i to n by i do p[j]:=0
    every i:=2 to n do if p[i]=1 then write(i)
    end
```

Example. Here is a program to write out Fibonacci numbers using a queue to keep track of successive numbers. Compare to Figure 11 on page 50 and Figure 12 on page 51.

```
    procedure main()
    local f,i,n
    n:=100000
    f:=[1,1]
    repeat {
        i:=get(f)
        if i>n then break
        write(i)
        put(f,i+f[1])
    }
    end
```

9.5 Other list functions

Table 37 shows some further functions that apply to lists:

Table 37 Other built-in list functions

`copy(L)`	creates a new list with the same contents as list L.
`image(L)`	`"list_`*num*`(`*leng*`)"`. A count is incremented each time a list is created. *num* is the value of the count when this list was created. *leng* is the list's current length.
`set(L)`	creates a set whose initial contents are the elements of the list L. See Chapter 11 on page 117.
`sort(L)`	creates a new list whose contents are the elements of list *L* in sorted order. Elements of the same type are grouped together. Lists, records, and other mutable objects are sorted in their group by their order of creation.
`sortf(L,i)`	creates a new list whose contents are the elements of list *L* in sorted order. Records and lists contained in *L* with a size of at least *i* are sorted by their *i*th field.
`type(L)`	`"list"` if `L` is a list.

Chapter 10 Tables

10.1 Creation, lookup and assignment

A table lets you associate values with keys. Both the keys and values can be of any type. Values may be looked up by the key they are associated with. A given key can have at most one value in the table, but the same value may be associated with any number of keys.

You create a table by calling the `table` function:

```
t:=table( )
```

or

```
t:=table(x)
```

Both create empty tables. They differ in what value you find when you look up a key that is not defined.

You assign a value, v, to a key, k, in the same fashion you would assign to an element of a list:

```
t[k]:=v
```

You look up the value associated with key k again using the same syntax as subscripting:

```
t[k]
```

Having assigned `t[k]:=v`, when we look up `t[k]` we get `v`. Actually, we get a variable containing the value `v`. We can assign another value to it.

Suppose `y` is not a key in table `t`. What happens when we look up `y`? That depends on how we created the table. If we used `t:=table( )`, `t[y]` gives us a variable containing `&null`. If we used `t:=table(x)`, `t[y]` gives us a variable containing the value `x`.

10.2 Initial value &null, \ and / idioms

When we create a table with a null initial value:

```
t:=table( )
```

we can check for a key being defined using the \ or / unary operator. Form

`\t[x]` will succeed if key x is defined and fail if it is undefined. Form `/t[x]` is just the other way around.

Suppose you want to map keys into lists. You do not want to say

```
t:=table([])
```

because all new keys would be assigned the same list. What you wish to do is create a new list whenever a new key is used:

```
/t[x]:=[]
```

For example, suppose you wish to make a table, t, that will map each character that occurs in a string, s, into a list of the positions at which it occurs. You might use the following code:

```
t:=table()
every i:=1 to *s & x:=s[i] do {
     /t[x]:=[]
     put(t[x],i)
}
```

Here is a technique (suggested by Todd Proebsting) to assign objects unique IDs:

```
/t[x] := "tmp" || *t
```

10.3 Other initial values

The most common use for creating a table with a non-null default value is to count the number of occurrences of items. Just make the default value zero, and whenever you find an item, increment its count. For example, to count the number of occurrences of characters in a string, s, you can use:

```
t:=table(0)
every t[!s]+:=1
```

10.4 Sort

You create a sorted list of the contents of a table using procedure sort. The call is

```
sort(t,k)
```

where k is an integer from 1 to 4.

Your options are:

- the items can be sorted by the key or the value. If k is odd, Icon sorts by key. If k is even, it sorts by value.
- the key/value pairs can come out in two elements sublists [...,[key,value], [key,value],...], or they can alternate in the top level list [...,key, value, key, value,...]. If k is 1 or 2, Icon gives you the two-item sublist form. If k is 3 or 4, it alternate the keys and values.

You do not get to choose ascending or descending order. They come out in ascending order. Just go through the list backwards if you want descending order.

Here are two ways to count the occurrences of characters in a string and write out character counts:

Using the key/value pairs:

```
t:=table(0)
every t[!s]+:=1
every p:=!sort(t,1) do
 write(image(p[1]),"\t",p[2])
```

(We use `image` to print out a representation of non-printing characters.)

Using the alternating keys and values:

```
t:=table(0)
every t[!s]+:=1
L:= sort(t,3)
while write(image(get(L)),"\t",get(L))
```

10.5 Generating keys and values

The unary ! operator will generate all the values in the table as variables. More useful is the function key.

```
key(t)
```

generates all the keys in the table.

If you would like to generate all the key/value pairs in a table, t, as two-elements lists, but you don't want to sort the table, you could use a subexpression like:

```
every .... p:=[k:=key(t),t[k]] ...
```

10.6 Functions

Here are some functions that operate on tables:

Table 38 Functions that apply to tables

`copy(T)`	returns a copy of table T
`delete(T,x)`	removes the key x and its value from table T.
`image(T)`	returns a string `"table_`*num*`(`*size*`)"` Icon counts the number of tables created and remembers the number of each table. *num* is the number of the table. *size* is the number of key/value pairs in the table.
`insert(T,x,y)`	same as T[x]:=y

Table 38 Functions that apply to tables

`key(T)`	generates all the keys in the table
`member(T,x)`	succeeds if key x is in table T. Returns x if it succeeds.
`sort(T)`	same as `sort(T,1)`
`sort(T,i)`	returns a list containing the keys and values from table T. If ki is the ith key and vi is its corresponding value, the resulting list is: [[k1,v1],[k2,v2],...[kn,vn]] sorted by keys if i=1 [[k1,v1],[k2,v2],...[kn,vn]] sorted by values if i=2 [k1,v1,k2,v2,...kn,vn] sorted by keys if i=3 [k1,v1,k2,v2,...kn,vn] sorted by values if i=4
`table( )`	returns a new table. Attempting to look up a key not in the table returns `&null`. For example, `t[[]]` will yield `&null` because the new list `[]` can't be in the table.
`table(x)`	returns a new table. Attempting to look up a key not in the table returns the value of `x`. For example, `t[[]]` will yield the value of `x` because the new list `[]` cannot be in the table. The expression `x` is evaluated when the table is created, so if you execute `t:=table([])`, all new keys you look up will point to the *same* list.
`type(T)`	returns `"table"`

10.7 Table operators

The unary *, !, and ? operators apply to tables as they do to other structures:

Table 39 Table operators

`* t`	returns the size of the table
`? t`	returns a random value in the table as a variable.
`! t`	generates the values in the table, as variables. The function `key(t)` is usually more useful.

10.8 Example: word count

Here is a program to count the number of occurrences of words in the input file. It is composed of a procedure, `word`, and a main procedure.

The procedure `word` will generate lists of length two—the identifiers in the input up paired with the numbers of the lines they occur in. (We will use the same procedure in a cross-reference program later, which needs the line numbers.) It uses the procedure `idents` to generate identifiers in a string in Figure 23 on page 75.

Figure 29 Word: Generating the words in a file

```
procedure word()
local line,s,lno,i,j
lno:=0
while line:=read() do {
     lno+:=1
     suspend [idents(line),lno]
}
fail
end
```

The main procedure uses a table of words with a default value of zero to simplify incrementing counts. The words are sorted before being written out.

Figure 30 Count occurrences of words in the input

```
procedure main()
local w,h
h := table(0)
every w:=word() do {
     h[w[1]] +:= 1
}
h := sort(h,3)
while write(get(h),"\t",get(h))
end
```


Chapter 11 Sets

11.1 Creation

A set is an unordered collection of objects without duplication.

You create a set by calling the `set` function:

```
set( )
```

or

```
set(L)
```

If you do not pass a parameter, the set function creates an empty set. If you pass it a list, it will create a set containing all the elements of the list (omitting duplicates).

11.2 Operators

The three unary operators that apply to all structured types apply, of course, to sets—the size operator, *, the element generator, !, and the random selection, ?. The identity test operator, === (and its complement,~===), will succeed (or fail) if the two operands are the same object so that any changes to one will be seen through the other.

The same set union (++), intersection (**), and difference (--) operators that apply to csets also apply to sets, but there is no complement operator.

Table 40 Set operators

operator	precedence	means
`* s`	12	returns the size of the set (number of elements)
`! s`	12	generates all the elements of the set
`? s`	12	chooses a random element of the set (return it, but do not remove it from the set)
`s1 ** s2`	9	intersection: creates a new set containing those elements that are in both s1 and s2
`s1 ++ s2`	8	union: creates a new set containing those elements that are in either s1 or s2 or both

Table 40 Set operators

operator	precedence	means
`s1 -- s2`	8	difference: creates a new set containing those elements that are in s1 but not in s2
`s1 === s2`	6	if s1 and s2 reference the same set, === succeeds returning that set, but fails otherwise
`s1 ~=== s2`	6	if s1 and s2 reference different objects, ~=== succeeds returning that s2, but fails otherwise

11.3 Functions

Elements may be added to sets with function `insert`, removed with `delete`. Function `member` tests to see if an element is a member of a set. The functions `copy`, `image`, and `type`, that apply to all types, of course, apply to sets as well.

Table 41 Functions that apply to sets

`copy(S)`	creates a copy of set S
`delete(S,x)`	deletes element x from set S. Returns set S.
`image(S)`	returns a string `"set_`*num*`(`*size*`)"` Icon counts the number of sets created and remembers the number of each set. *num* is the number of the set. *size* is the number of members in the set.
`insert(S,x)`	inserts element x into set S (if it is not already present). Returns S.
`member(S,x)`	succeeds if x is a member of set S, fails otherwise. Returns x if it succeeds.
`set()`	creates an empty set.
`set(L)`	creates a set composed of the elements of list L.
`sort(S)`	creates a list composed of the members of set S in sorted order. Elements of the same type are grouped together. Lists, records, and other mutable objects are sorted in their group by their order of creation.
`sortf(S,i)`	creates a new list whose contents are the elements of set *S* in sorted order. Records and lists contained in *S* with a size of at least *i* are sorted by their *i*th field.
`type(S)`	returns `"set"`

11.4 Idiom: to-do sets

One use of sets is to keep track of things that have to be processed or that already have been processed. Even though you may discover several times that something should be processed, you still want to process it only once. If you try keeping a to-do list, you might put the same item on several times. With sets, you do not have duplicates.

11.5 Examples using sets

11.5.1 Cross reference

Here is a program to report the identifiers that occur in a file (the standard input) and lines in which they occur. It keeps sets of line numbers so that a line will be reported at most once for an identifier. On output, the identifiers and line numbers are sorted. This code uses the procedure word defined in Figure 29 on page 115.

Figure 31 Cross reference listing

```
procedure main()
local w,xr,lnos
xr := table()
every w:=word() do {
     /xr[w[1]] := set()
     insert(xr[w[1]],w[2])
}
xr := sort(xr,3)
while writes(get(xr),"\t") & lnos:=get(xr) do {
     every writes(" ",!sort(lnos))
     write()
}
end
```

11.5.2 Cross reference without reserved words

Suppose we want the cross reference listing to omit Icon reserved words. We can construct a set of those words and check to see if the word we have found is in the set before putting it in the table.

Figure 32 Cross references without reserved words

```
procedure main()
local w,xr,lnos, reserved
xr := table()
reserved:=set(["global","local","static","record",
     "procedure","end","initial","re-
turn","fail","suspend",
     "if","then","else","case","of",
     "every","do","while","until",
     "repeat","break","next",
     "to","by","not"])
```



```
every w:=word() & not member(reserved,w[1]) do {
     /xr[w[1]] := set()
     insert(xr[w[1]],w[2])
}
xr := sort(xr,3)
while writes(get(xr),"\t") & lnos:=get(xr) do {
     every writes(" ",!sort(lnos))
     write()
}
end
```

11.5.3 Eight queens problem

The eight queens problem is to place eight queens on a chess board (8×8) so that no two attack each other. That means no two queens may be on the same column, row, or diagonal.

The work is done in procedure placeQueen(c) that places a queen in each position of column c where it is not on an occupied row or diagonal and then calls placeQueen(c+1) to place the next queen. When called with c>8, placeQueen will call writeBoard to write out the configuration.

Sets are used to indicate which rows and diagonals are occupied. The trick for representing diagonals is as follows:

Number the rows from 1 to 8 from top to bottom and the columns from 1 to 8 left to right.

Notice that the square at row r and column c is on two diagonals, one going up to the right and the other going down.

For every square on the diagonal going down to the right, the square's row number, i, and column number, j, have the same difference: i-j = r-c.

For all squares on the diagonal going up to the right, the sum of the row number, i, and column number, j, are the same: i+j = r+c.

So to check whether square (r,c) is attacked by some other queen, we check to see if r is a member of set row, r-c is a member of set diffDiag, and r+c is a member of sumDiag.

Figure 33 The eight queens problem

```
global numQueens, row, sumDiag, diffDiag, placement

procedure main()
numQueens:=8
row:=set()
sumDiag:=set()
diffDiag:=set()
placement:=[]
placeQueen(1)
```



```
end

procedure placeQueen(c)
if c>numQueens then
     return writeBoard()

every r:=1 to numQueens &
     not member(row,r) &
     not member(sumDiag,r+c) &
     not member(diffDiag,r-c) do {
          insert(row,r)
          insert(sumDiag,r+c)
          insert(diffDiag,r-c)
          put(placement,r)
          placeQueen(c+1)
          delete(row,r)
          delete(sumDiag,r+c)
          delete(diffDiag,r-c)
          pull(placement,r)
}
return
end

procedure writeBoard()
local board,c,r
board:=list(numQueens,repl("_",numQueens))
everyr:=1 to numQueens &
     c:=1 to numQueens &
     (r+c)%2=1 do
          board[r][c]:="x"

every c:= 1 to numQueens do {
     r:=placement[c]
     board[r][c]:="Q"
}
every write(!board)
write()
return
end
```

11.5.4 Primes sieve using sets

Here is the primes sieve program again, this time using sets rather than bits, strings, or lists. For the other versions, see Figure 26 on page 87 and Figure 24 on page 75.

```
procedure main()
local p,i,j,n
n:=1000
p:=set()
```



```
every i:=2 to n do insert(p,i)
every i:=2 to sqrt(n) &
     member(p,i) &
     j:= i+i to n by i do delete(p,j)
every i:=2 to n & member(p,i) do write(i)
end
```


Chapter 12 Records

12.1 Record declarations

Record types in Icon are like *struct*'s in C or records in Pascal. They are declared at the same level as procedures and globals:

```
record recordName(field1, field2,..., fieldn)
```

The `recordName` is the name of the record type. It also becomes the name of a record constructor procedure. The fields are identifiers that name the members or fields of the record. The list of fields may be empty.

12.2 Creation

You create an instance of a record by calling its record constructor procedure passing it initial values for the fields, e.g.,

```
r := recordName(e1,e2,...,em)
```

If you pass more values than there are fields, the rightmost arguments are ignored. If you pass fewer, the remaining fields are initialized to `&null`.

Records are mutable values. They are accessed via pointers.

12.3 Field access r.f

The normal way to access a field uses the dot notation:

```
r.f
```

will select field f of record r. It is a variable; you can assign to it. More than one record type can have the same field name; Icon selects the correct field at run time. If the record does not have a field with that name, Icon reports an error at run time.

12.4 Generating fields: ! (unary)

The unary exclamation point operator will generate the fields in a record from left to right. This is used for displaying the contents of a record in debugging routines and for persistence, saving data structures in string format on disk.

12.5 Subscripting records: r["f"] r[i]

Two other ways to access a record's fields are to subscript it as if it were a table

or a list.

If you subscript a record with a string containing a field name, you will select the field of the record with that name. If the record does not have a field by that name, the attempt to subscript fails, rather than causing a run-time error.

If you subscript a record with an integer index, you get the field at that position.

12.6 Applying a procedure to the fields: ! (binary)

You can use the binary operator `!` to apply a procedure to a record. The fields of the record are passed as parameters to the procedure.

You can also use the binary operator `!` to apply the record constructor to a list.

If you wanted to convert the fields in a record, `r`, to a list, you could use a procedure:

```
procedure returnArgs(L[])
return L
end
```

and call it with

```
returnArgs!r
```

12.7 Record operators

Here is a summary of the operators that apply to records:

Table 42 Operators that apply to records

`* r`	size (number of fields) or record `r`
`! r`	generate the fields of `r` from left to right as variables
`? r`	choose a field of `r`, randomly. (No, we don't have any idea what we would use it for either.)
`p ! r`	pass the fields of `r` to procedure `p`
`rc ! L`	create a record by passing the fields of structure (probably list) `L` to record constructor `rc`.
`r . f`	select field `f` of record `r` (a variable)
`r [ i ]`	select the `i`th field of record `r` (a variable), or fail if there is no such field
`r [ s ]`	select field of record `r` with the name given by string `s` (a variable), or fail if there is no such field

12.8 Record functions

There are three built-in functions that particularly apply to records:

Table 43 Functions that apply to records

`copy(r)`	creates a new record with the type and contents of record `r`.
`image(r)`	returns a string `"t_i(l)"` where t is the record type, `i` is the number of this record, and `l` is the length.
`type(r)`	returns the type of record `r` as a string, e.g., if r was created `r:=R(...)` then `type(r)=="R"`.

Chapter 13 Data Types and Conversions

13.1 Variables and Values

In Icon, variables correspond to storage cells into which values can be assigned and from which values can be fetched. Expressions can yield variables, but variables are not quite "first class" objects. Here are some of the rules for variables:

- You can declare variables with global, local and static declarations.
- Procedure declarations create global variables with the name of the procedure which are initialized to the code for the procedure. You can assign another value to the variable, losing access to the procedure.
- Some keywords are variables. Some are not.
- Subscripted lists, tables, and records are variables; however, a list subscripted with a range, *e.g.*, L[i:j], is a value, not a variable.
- A subscripted variable that contains a string is a variable; but it can only be assigned a string.
- A subscripted string value is a value.
- A procedure can return or suspend yielding a variable, but a local or static variable will have its value returned instead (locals will not be there any more once the procedure has returned).
- The unary operators ! and ? applied to lists, tables, or records yield variables. They also return variables when applied to variables containing strings.
- The unary / and \ operators preserve variables (i.e., if they're given a variable, they return it), although they examine the values the variables contain.
- The assignment operators require variables on the left hand side. They return their left hand sides as variables. The exchange operators require variables on both sides; they also return the left hand side variable.
- The &, |, if and case control constructs preserve variables.
- The unary dereference operator, `.`, returns the value of x. It is used to force the value to be fetched immediately.
- You *cannot* assign a variable to a variable and "go indirect through it" to access the cell it points to.

- A variable is left as a variable until its value is needed.
- Variables in a parameter list are left as variables until the procedure is called, meaning that assignments to the variables later in the parameter list will change the value passed. The same is true of variables within the bracketed list constructor, [e1, e2, ..., en].

Table 44 Operations on variables

`. x`	returns the value of x. It is only significant when x is a variable. Normally a variable is left as a variable until its value is needed. For example, variables in a parameter list are left as variables until the procedure is called, meaning that assignments to the variables later in the parameter list will change the value passed. The `.` (dereference) operator will force the value to be fetched immediately.
`variable(s)`	returns the variable or variable keyword whose name is contained in string s. It will only return a variable known at the place of call—you can only access a local variable within a procedure.

13.2 Operations On Arbitrary Types

Most Icon operators restrict the types of their operands, but a few work with any type.

The assignment operators will assign any type of value to a variable, although only a string can be assigned to a subscripted string variable. There are two varieties of assignment operators, irreversible and reversible. The irreversible assignment operators, := and :=:, simply perform the assignment and are done with it. The reversible assignments, <- and <->, will be resumed during backtracking and restore the value the variable(s) had before the assignment.

The object equal operators (=== and ~===) test two objects to see if they are identical. Structured objects, lists, tables, records and such, may have identical contents, but they are only equal if they are the same object.

Table 45 Operations on arbitrary types

operator	precedence	means
`x === y`	6	succeed if the values of x and y are the same object, or if they are immutable objects like numbers, strings, or csets, succeed if they have the same contents. Otherwise, fail.

Table 45 Operations on arbitrary types

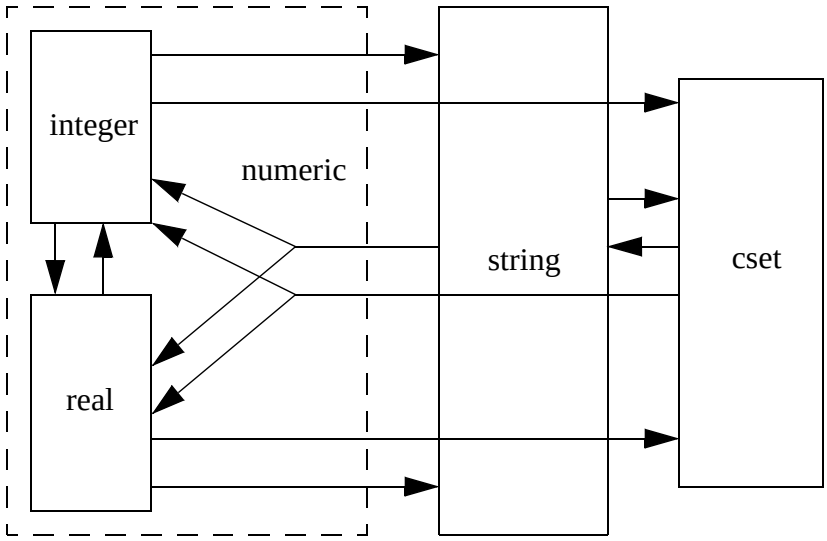
operator	precedence	means
`x ~=== y`	6	succeed if the values of x and y are different mutable objects, or if they are immutable objects like numbers, strings, or csets, will succeed if they have different contents. Otherwise, fail.
`x := y`	3	assign the value `y` and return variable `x`.
`x :=: y`	3	exchange the values in the variables `x` and `y` and return variable `x`.
`x <- y`	3	assign the value `y` and return variable `x`. If resumed during backtracking, restore the value x had before the assignment.
`x <-> y`	3	exchange the values in the variables `x` and `y` and return variable `x`. If resumed during backtracking, restore the values x and y had before the assignment.
`x | y`	5	yields the sequence generated by the left operand followed by sequence generated by the right. (It is not a true operator.)
`x & y`	1	for every element of the sequence generated by the left operand, it yields the sequence generated by the right.
`copy(x)`		`copy(x)` creates a new instance of any mutable object `x` (list, table, set, record) that has the same internal structure as `x`, but is not equal (`~===`) to `x`. Immutable objects like numbers, strings, csets are left as is.
`image(x)`		`image(x)` creates a legible string that in some sense describes object x. If x is a string or a cset, the image has quotes around it and has its illegible characters represented by escape sequences.
`type(x)`		`type(x)` returns a string that represents the type of object x.

13.3 Built-in conversions

The built-in data conversions in Icon involve integer, real, string, and cset values. In a context where one of those types is required, Icon will attempt to convert a value of any of the other types to that type. However, some of the

conversions go through an intermediate type. The conversions are shown in Figure 34.

Figure 34 Built-in data conversions.



Integers and reals are converted directly to each other and directly to string. To convert an integer or real to a cset, it is first converted to a string and then the characters in its string representation are converted to a cset. That is the meaning of the arrows going from integer and real to cset going through the box labeled string.

Conversions from string to integer or real first go into either *numeric* representation, meaning the converted string could be integer or real—it all depends on which type of value the string represents. If either type would do, the number is left as which ever type the string represented. If a specific type is required, then an integer is converted to real, or real to integer, as required.

A string is converted into a cset by making a set of all the characters in the string. A cset is converted to a string by simply listing the characters in it in order. A cset can be converted to an integer or real if its sorted characters form a valid numeric representation.

13.4 Translating structures to strings

The Icon Program Library contains two procedures, `encode` and `decode`, to translate an arbitrary data structure into a string and to translate the string back into the data structure. In the graph of the data structure multiple paths to the same object are permitted, cycles are permitted. The structure will be recreated when the string is converted back.These procedures are contained in file `code-`

`obj.icn` and are described in Table 46.

Table 46 Encoding and decoding data structures.

`decode(x)`	translates a string produced by encode back into a data structure isomorphic to the one encoded. `link codeobj`
`encode(x)`	translates the data structure accessible from x into a string and returns that string. Any contained files, functions, procedures, co-expressions, and windows cannot be properly contained in a string, so don't try to include them. `link codeobj`
`xdecode(f)` `xdecode(f,p)`	reads, reconstructs, and returns the Icon data structure from file `f` that was previously saved there by xencode. Files, co-expressions, and windows are decoded as empty lists (except for files `&input`, `&output`, and `&errout`). Fails if the file is not in xcode format or if it contains an undeclared record. If p is provided, `xdecode` reads the lines calling `p(f)` rather than `read(f)`. See `xencode` for an idea of what to use this for. `link xcode`

`xencode(x,f)`
`xencode(x,f,p)`
	encodes and writes the data structure `x` into file `f`. The data structure can be read back in by `xdecode`. If parameter p is provided, it is called in place of `write`, *i.e.* `p(f,...)` instead of `write(f,...)`, in which case `f` need not be a file, *e.g.*
	`xencode(x,L:=[],put)`
	will encode the data structure into a list, `L`.
	
	`link xcode`

Procedure `encode` produces strings without control characters or newlines in them, so the strings can safely be written out to files and read back in. However, if the data structures are large, you would do better to write them and read them back with procedures `xencode` and `xdecode` from file `xcode.icn`.

Procedures `xencode` and `xdecode` can also be used, as shown in the table, to linearize data structures into a list of strings.

Chapter 14 Debugging

14.1 Basic debugging

You will do your debugging of Icon programs at run-time.

Because Icon is a typeless language, the translator cannot check that you are matching operand types properly for operators. Mismatched types will cause your most common run-time error. Fortunately, when Icon detects an error, it will stop execution and inform you of the problem. It will give the file and line number where the error occurred, a trace-back of all the active procedure calls, and an explanation of what was wrong.

For problems with your algorithms, however, there is less help. There is no "Icon Programming Environment" to allow you to single-step your program or put in breakpoints and watch expressions. You debug your programs either by turning on tracing or by inserting output commands in the code.

Tracing is controlled by the variable keyword `&trace`. When `&trace` is non-zero, procedure calls, returns, suspends and resumes, result in a line being written to the standard error output. The value of `&trace` is decremented each time a line is written, so if you set it to a positive value, it will only trace until that value goes to zero. You can assign a value to `&trace` to start tracing. You can stop the tracing by assigning `&trace := 0`. If you assign `&trace:=-1`, tracing will continue indefinitely.

You can initialize the value of `&trace` before running the program by assigning a value to the environment variable TRACE, e.g.,

```
set TRACE=1000
```

or

```
setenv TRACE 1000
```

If you need to examine the state of the computation at some point within the program, the easiest output command to add is a call to the `display()` function. It will show the active procedures and the contents of variables.

If you want to tailor your own debugging messages, you probably want to use functions `image(x)` and `name(x)` to get strings that represent values and variables.

Here are some functions and keywords you may find useful.

Table 47 Debugging functions and keywords

`display(i,f)`	writes to file f the names of the i most recently called active procedures, their local variables, and the global variables. Used for debugging.
`display(i)`	writes to `&errout` the names of the i most recently called active procedures, their local variables, and the global variables. Used for debugging.
`display()`	writes to `&errout` the names of all the active procedures, their local variables, and the global variables. Used for debugging.
`&dump`	a variable. If `&dump` is nonzero on program termination, Icon calls `display()` before terminating.
`&error`	controls whether errors cause the program to terminate. When zero, an error causes program termination with an error message. If nonzero, an error causes a failure and `&error` is decremented. The error message that would have been reported is instead assigned to keywords `&errornumber`, `&errortext`, and `&errorvalue`.
`errorclear()`	clears the indication that an error has occurred. References to the keywords `&errornumber`, `&errortext`, and `&errorvalue` fail until the next error has occurred.
`&errornumber`	the number of an error.
`&errortext`	the text explaining the error.
`&errorvalue`	the offending value (e.g., whose type didn't match). Access to `&errorvalue` will fail if there is no offending value associated with the error.
`&errout`	the standard error output file.
`&file`	the file name of the file this code was compiled from.
`&host`	the name of the computer system the program is running on.
`image(x)`	returns a legible representation of object x. For details, see Section 6.6, String editing and conversion functions, on page 65.
`&level`	is the number of levels of active procedures calls.
`&line`	is the number of the line this keyword occurs on.

Table 47 Debugging functions and keywords

`name(x)`	just as `image(x)` gives a legible indication of a value, `name(x)` gives a legible indication of a variable. If the variable, `x`, is a keyword or declared variable, name gives its name as a character string. If it is a component of a structure, `name(x)` gives the structure type (`list`, *record*,...) and the way the component is usually accessed, e.g., "`list[2]`", "`rec.f`".
`&progname`	the file name of the executing program. It's a variable. You can assign another string to it if you wish.
`runerr(i,x)`	cause the program to terminate with a standard run time error message for error number `i` and offending object `x`.
`&trace`	when not equal to zero, every procedure call, return, suspension, or resumption writes a message to `&errout` and decrements `&trace`.
`type(x)`	returns the type of the value x as a string. For details, see section 6.6 on page 65.
`&version`	is a string representation of the version of Icon that is executing.

The current numbers and messages for Icon's run-time errors are shown in Table 48 on page 135.

Table 48 Run-time errors

number	message
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

Table 48 Run-time errors

number	message
111	variable expected
112	invalid type to size operation
113	invalid type to random operation
114	invalid type to subscript operation
115	structure expected *(e.g. list, set, or table)*
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
127	improper event monitoring setup
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
201	division by zero
202	taking the remainder of division by zero
203	integer overflow

Table 48 Run-time errors

number	message
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 argument 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
316	interpreter stack too large
318	co-expression stack too large
401	co-expressions not implemented
402	program not compiled with debugging option
500	program malfunction
600	vidget usage error

14.2 Monitoring storage

Because Icon automatically reclaims storage (called "garbage collection"), the amounts of storage used by programs can cause significant differences in performance. Icon provides some functions and keywords to allow you to see what the storage requirements of your program are.

There are three sections of storage:

1. the static section composed of blocks that are never moved,
2. the string region filled with characters,
3. the block region, composed of blocks of storage that are allocated for most structured data types.

Active storage in both the string and block region is compressed to one end of their area when they are garbage collected.

The garbage collector only runs when storage in some region is exhausted or when you call the function collect. Since the program does not run concurrently with the garbage collector, you may notice your program pausing every so often. Do not use Icon for real-time systems.

If you are holding onto a lot of storage in some region—that is, if it is all accessible all the time—then the garbage collector may run frequently collecting only a little space each time. If your program is about to run out of space, it may take a very long time to actually do so.

If you suspect you are having problems with storage, you can use the function and keywords shown in the following table to find out.

Table 49 Storage management

`collect( )`	forces a garbage collection.
`collect(i)`	forces a garbage collection of region i, where i specifies the region 0—no specific region 1—static region 2—string region 3—block region
`collect(i,j)`	forces a garbage collection of region i, where i specifies the region as shown for `collect(i)` above. Fails if there are not at least j bytes available in the region after collection.

Table 49 Storage management

`&collections`	generates four values: • the number of garbage collections • the number caused by attempts to allocate in the static region • the number caused by attempts to allocate in the string region • the number caused by attempts to allocate in the block region. The first value may be larger than the sum of the other three due to calls to `collect()`.
`&regions`	generates the current sizes of the three regions: static, string, and block. The size of the static region may not mean anything: Icon might allocate more space from the system when needed.
`&storage`	generates the amount of space currently in use in the three regions: static, string, and block. Again, the space occupied in the static region may not mean anything.

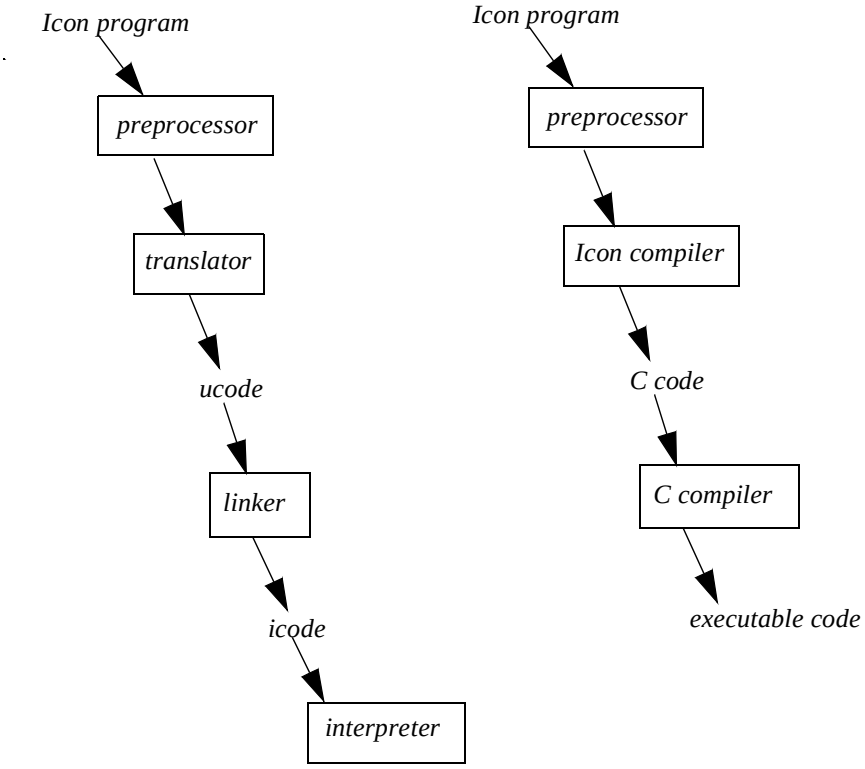
Chapter 15 Writing systems

15.1 Translator commands

15.1.1 Translator and compiler

Most people use the Icon translator and interpreter system as shown at the left in Figure 35. On some systems an Icon compiler is available. The compiler can

Figure 35 Icon translator and compiler.



produce a faster-running Icon program, but the compiler itself is slow and requires a great deal of memory to run. The compiler generates a C program which then must be compiled by a C compiler.

The command line to translate an Icon program is

```
icont options and file names
```


The command line to compile an Icon program is

```
iconc options and file names
```

The beta version of Icon translator with graphics facilities for Microsoft Windows is called `wicont`.

15.1.2 Translating multiple files

It's best to divide large programs into a collection of modules. Each module is placed in a different file and the files are combined to produce the overall program.

In Icon, you can compile multiple files together by the command

```
icont f1 f2 ... fn
```

where *f1 f2 ... fn* are the names of the files. The executable file produced takes its name from *f1*.

After you have some of the modules well debugged, you do not need to compile them over again each time. You can compile a file to be linked in later by the command

```
icont -c f2
```

which will produce a pair of files, *f2.u1* and *f2.u2*. You then include those files in a compilation by referring to file *f2.u*:

```
icont f1 f2.u ... fn
```

You can have the precompiled files included automatically, without listing them on the command line, by listing them in a module that uses them. Use the `link` declaration *inside* your icon program:

```
link f2,f3,...,fn
```

where each *fi* is either an Icon identifier that is the same as a file name (without the ".u") or a string that is a file name (probably because of some non-identifier characters). For example:

```
link array,"hash-tbl"
```

The *link* declaration causes Icon to include the compiled (ucode, *.u1* and *.u2*) files named. It will search for them through several directories. First it searches the current directory. If they are not found there, it reads the environment variable IPATH. IPATH should be set to a list of directory paths separated by blanks which Icon will investigate in order to find the named files.

15.1.3 Command-line arguments

Commands to the translator are shown in Table 50. These vary somewhat by the version of Icon, so you will need to consult the documentation on your version

if these don't work.

Table 50 . Command line flags for icont.

`-c`	translate the files into ucode files, not all the way to an interpretable or executable file. (Or, in the case of the compiler, to code in the C programming language.)
`-C` *`cc`*	in the Icon compiler, use the C compiler located at *cc*.
`-e` *`file`*	redirect error output to `file` rather than to the standard error output.
`-f` *`opts`*	enable the features specified by the option characters in the string *opts*. • a—all features. • d—enable debugging features. • e—enable error conversion. • l—enable large integer arithmetic. • n—keep track of source code line numbers and filenames. • s—enable string invocation. These features are disabled in the compiler in order to achieve greater execution speed. String invocation is also now disabled in the translator to decrease the size of the translated files. Instead of specifying `icont -fs ...`, it is better to include `invocable all` in your program.
`-n` *`opts`*	in the Icon compiler, disable the specific optimizations given by characters in *opts*. • a—all. • c—control flow optimizations other than switch statements. • e—expand operations in-line • s—switch statements. • t—type inference.
`-o` *`filename`*	name the translated Icon program `filename`, rather than taking its name from the first file listed in the `icont` command.
`-p` *`ccarg`*	in the compiler, pass argument *ccarg* to the C compiler.
`-r` *`path`*	use the run-time system located at *path*.
`-s`	suppress informative messages from the translator

Table 50 . Command line flags for icont.

`-t`	initialize `&trace` to `-1`, that is, run the program with tracing turned on.
`-u`	issue warnings for undeclared identifiers used in the program.
`-v` *`i`*	controls the verbosity of the translator's output: • -v 0 suppresses messages the same as -s • -v 1 is the default • -v 2 reports the sizes of sections of the icode file • -v 3 lists globals that are being discarded.

15.2 Global name space

Unfortunately, Icon has a single global name space. There are no variables or procedures private to files. This means you may have to deal with name collisions: using the same name in more than one module.

Probably the safest way to deal with this is to choose a name for each module and combine it with the local name using an underscore. For example, use `setHigh_Space` or `Space_setHigh` for the `setHigh` procedure provided in the `Space` module.

15.3 The preprocessor

Icon provides an inferior version of the C processor to aid in program development. The C preprocessor processes commands that begin with a #-sign and a directive name, but since # introduces comments in Icon, Icon begins its preprocessor directives with $. The directives are given in Table 51.

Table 51 Preprocessor directives.

`$define` *`name text # cmmt`*	defines *name*. When *name* is encountered subsequently in the program, it will be replaced with *text*. The replacement text itself is scanned for replacements, although Icon suppresses further replacements of *name* within *text* to avoid infinite loops. The *text* may be empty. The `# cmmt` is a comment, which may, of course, be omitted.
`$error` *`text`*	causes a fatal compile-time error displaying *text*. You would use it within `$ifdef`'s when you have determined the required options are not available.

Table 51 Preprocessor directives.

`$ifdef` *`name`*
`...lines-if-defined`
`$else`
`...lines-if-not-defined`
`$endif`
	if *`name`* is defined, include the *`lines-if-defined`*, otherwise include the *`lines-if-not-defined.`* The `$else` and `lines-if-not-defined` are optional.

`$ifndef` *`name`*
`...lines-if-not-defined`
`$else`
`...lines-if-defined`
`$endif`
	if *`name`* is defined, include the *`lines-if-defined`*, otherwise include the *`lines-if-not-defined.`* The `$else` and `lines-if-defined` are optional. These directives may be nested. They must be balanced, i.e. occur as a group, in order, in the same file—they cannot extend into or out of an included file.

`$include` *`filename`*	includes the contents of the specified file in place of the `$include` directive. If *`filename`* is not an Icon identifier, it must be quoted.
`$line` *`n`* `$line` *`n filename`*	tells the translator to consider this line to be number *n*. This differs from C, where the line command tells the compiler to consider the following line to be number *n*. If filename is supplied, the subsequent lines are considered to be in the specified file. If *`filename`* is not an Icon identifier, it must be quoted.
`$undef` *`name`*	undefine, i.e. remove the definition of, *name*.

15.4 Environment Inquiries

There are a number of functions and keywords that provide information to the Icon program. They allow an Icon program to find out about the files it was compiled from, the features of the version of Icon running, the host machine, the time, and the values of environment variables in the operating system.

Table 52 Environment inquiries

`&clock`	returns a string giving the current time of day `"hh:mm:ss"`, in 24 hour form.
`&date`	returns the current date in the form `"yyyy/mm/dd"`.
`&dateline`	returns the current date and time of day as a string.
`&features`	generates strings indicating the features of this version of Icon.

Table 52 Environment inquiries

`function()`	generates the names of Icon's built-in functions.
`&file`	contains the name of the file the current line was compiled from.
`getenv(s)`	returns the string associated with the environment variable named `s`. The environment variables are string-valued variables maintained by the operating system (typically, the shell program).
`&host`	contains a string identifying the computer system Icon is running on.
`&line`	contains the number of the current line in the file it was compiled from.
`&progname`	contains the file name of the executing program.
`system(s)`	executes the string `s` as a system (shell) command and returns the exit status (an integer) by calling the C function `system`. It is not available on all systems, but it is available on UNIX©.
`&time`	returns the number of milliseconds since the program started executing.
`&version`	contains a string identifying the version of Icon that is executing.

Chapter 16 Co-expressions

16.1 What are co-expressions?

You can understand Icon backtracking by imagining a single "point of control" moving left to right through an expression and then backtracking right to left. Co-expressions are a way to have more than one point of control. Icon moves only one point of control at a time, that of the current co-expression, but it can resume moving the others at any time.

Warning. Co-expressions are not present in all Icon implementations.

16.2 Creation: create e

You create a co-expression by the form:

```
c := create e
```

which yields a co-expression ready to evaluate expression e. The co-expression saves a copy of the local variables of the procedure that executed the create. Whenever expression e refers to any local variable of the surrounding procedure, it is referring to its own copy. Assignments to local variables will not be seen by the procedure that created the co-expression.

16.3 Activating a co-expression

Suppose we created a co-expression

```
c := create e
```

You hand control to a co-expression, c, by using the form:

```
@ c
```

The co-expression will be allowed to execute until it passes control to another co-expression.

Typically, expression e in the co-expression c will execute until it generates a value. The value is then passed back as the value of the @c operation. When entered again, expression e will backtrack to generate another value. When e finally fails, the @c will fail.

When control is moving forward, the operation @c will *activate*, resume executing, the co-expression. It will not activate c during backtracking. For example


```
c:=create 1 to 3
while i:=@c do writes(i)
d:=create 1 to 3
every i:=@d do writes(i)
```

writes out

```
1231
```

Three times the `while` loop reactivates the co-expression, `c`, and it generates a new integer, and the fourth time it fails, leaving the loop. The `every`, however, activates the co-expression, `d`, generating the value 1. After writing out the 1, it backs into @d, which does not reactivate the co-expression, and control falls out of the loop.

16.4 States of a co-expression

Only one co-expression will be executing at a time. Any others will be waiting to execute in any of three states:

It may be waiting to begin execution, immediately following a `create` or a refresh, `^.`

It may have generated a value and be waiting to backtrack into its expression.

It may have explicitly given control to another co-expression with an @ operator and be waiting to have control passed back to it.

16.5 Getting the number of values generated

The size operator,

```
*c
```

will tell how many values a co-expression, `c`, has generated.

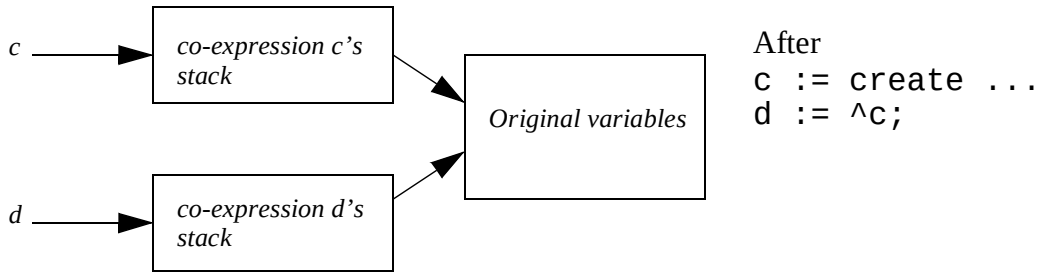
16.6 Refreshed copies

You can create a refreshed copy of a co-expression using the unary `^` operator. A co-expression saves a copy of its initial state, that is, the local variables of the procedure that executed the create and the address of the code it is to execute.

```
^ c
```

creates a new co-expression in the initial state `c` was in. It will start executing from the beginning. Figure 36 shows the internal data structure for co-expressions. The co-expression proper has a stack and a pointer to a shared block of storage with a snapshot of the original contents of the surrounding procedure's local variables. When a refreshed copy of a co-expression is created, a new stack is allocated and initialized from the snapshot.

Figure 36 Refreshed copies of co-expressions.



16.7 Symmetric activation: val @ c

A co-expression may explicitly pass control to another waiting co-expression no matter what state it is in. The general form for passing control is the binary @ operator.

```
e @ c
```

will pass the value of `e` to the co-expression `c`. If `c` is waiting at an `@` operator, the value passed to it will become the value of `c`'s `@` operation. If `c` is waiting either to begin execution or to backtrack, it is unable to receive the value, so the value is ignored. The unary operation, `@ c`, means the same as `&null @ c`.

16.8 Co-expression keywords

There are three keywords that are bound to co-expressions. They are shown in Table 53.

Table 53 Co-expression keywords.

`&current`	the currently executing co-expression.
`&main`	the co-expression that initially began executing the program.
`&source`	the co-expression that activated the current co-expression.

16.9 p { ... }

A procedure call,

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

is equivalent to

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


Chapter 17 Windows and Graphics

Warning: Much of this material was tested with a beta version of Icon for Windows 3.1. It may change. Windows and graphics are a relatively new feature of Icon. They are not available in all versions or implementations of Icon, and they may be subject to change. They are implemented under X-windows.

17.1 Windows

You can create windows in Icon to use for interaction with a user at a video terminal. Windows are objects of data type `window`. You create a window by opening it. A window has a number of attributes, some of which you can set to new values. You can read and write text through windows. You can draw two-dimensional figures in windows. You can receive events, e.g. mouse clicks, through windows.

Some of the graphics functions are implemented internally within the Icon runtime system. Others are implemented in Icon and must be linked in from the graphics section of the Icon Program Library. If you are using windows, include

```
link graphics
```

declaration in your program.

You create and open a window with the function WOpen. WOpen returns a window object. The parameters to WOpen are strings that assign values to window attributes. Each attribute assignment is a string of the form "*attribute_name=value*". Typically the minimum attributes that would be specified when opening a window would be the width and height of the window, although a window defaults to enough space for twelve lines of eighty columns of text.

Table 54 Common window attributes for WOpen.

`"bg=`*color*`"`	selects the background color for the window. The default is `"bg=white".` Colors are discussed in Section 17.6 on page 168.
`"fg=`*color*`"`	selects the foreground color for the window. The default is `"bg=black".` Colors are discussed in Section 17.6 on page 168.
`"height=`*h*`"`	height of the window in pixels. Defaults to enough for 12 lines of text.

Table 54 Common window attributes for WOpen.

`"pos=`*x*`,`*y*`"`	equivalent to: `"posx=`*x*`","posy=`*y*`".` Defaults to `"pos=0,0"`, i.e. the window is drawn at the upper left corner of the screen.
`"posx=`*x*`"`	the horizontal position of the left edge of the window on the screen.
`"posy=`*y*`"`	the vertical position of the top edge of the window on the screen.
`"size=`*w*`,`*h*`"`	equivalent to:`"width=`*w*`","height=`*h*`"`
`"width=`*w*`"`	width of the window in pixels. Defaults to enough for 80 columns of text.

The functions that operate on windows take the window object as their first parameter. However, the window parameter is optional. If it is omitted, the function uses the value of the keyword `&window`.

Keyword `&window` initially has the value `&null`. If `&window` has the value `&null`, then WOpen will assign it the newly created window. *If you only intend to use one window, you never need to use the window parameter to the graphics functions.*

Procedure WClose will close a window. WDone will wait until a Q or q character is typed in the window and then close it—Q being an Icon convention for "quit."

Table 55 Functions to open and close windows.

`WAttrib( W,s1,s2,...)`	sets and queries the attributes of a window. Each string is either `"`*`name`*`"` or `"`*`name=value`*`"` where name is the name of an attribute and value is a string representation of a value for that attribute. First WAttrib will perform assignments for all the `"`*`name=value`*`"` parameters, then it will generate the values for all the attributes named, left to right. Generates the values of the attributes. Fails on an attempt to set an invalid `bg`, `fg`, `font`, or `pattern`. Gives a run-time error on any other invalid value or name.
`WClose(W)`	closes the window W. The window is removed from the screen. Closing `&window` sets `&window` to `&null`.
`WDone(W)`	waits until a `Q` (or `q`) is typed in the window, then closes it.

Table 55 Functions to open and close windows.

`WOpen(` `s1,s2,...,sn)`	opens a new window with the values of its attributes given by the strings. Each string is of the same form as a value assignment in `WAttrib`: `"`*`name=value`*`"` where name is the name of an attribute and value is a string representation of a value for that attribute. Returns the window. Assigns the window to `&window` if `&window` is previously `&null`.

You can set attributes after a window has been opened by calling WAttrib and specifying the new attribute assignments as its parameters. You can also examine the current values of a windows attributes by calling WAttrib with just one string parameter giving the name of the attribute; WAttrib will return the current value of that attribute.

17.2 Graphics

17.2.1 Co-ordinates and angles

The coordinates are in pixels from the upper left of the window. The upper left has the x,y coordinates (0,0). The x coordinate increases to the right; the y coordinate increases downwards. This differs from the convention of graphing mathematical functions where the y coordinate would grow upwards.

If you wish to have the (0,0) point be somewhere else than the upper left, you can set the window attributes dx and dy. Attribute dx will be automatically added to any x coordinate to calculate the window-relative coordinate, and dy will be added to the y coordinate. For example,

```
WOpen("size=200,200","dx=100","dy=100")
```

will open a 200 by 200 pixel window, but position (0,0) will appear to be in the center of it.

Angles are measured in radians. When used as parameters to functions, angle 0 is horizontal to the right and angles increase *clockwise* from there.

17.3 Lines

17.3.1 Line-drawing functions

Icon provides a collection of functions to draw points, lines, curves, and closed figures in a window. The functions take a flat series of x and y coordinates in the window. By "flat" we mean that a point is not represented by a list of two coordinates, nor a line by a list of two points, but rather the x and y coordinates of the endpoints of a line are all at the top level in the function's parameter list. You will probably find that you will represent points and lines internally as structured objects and then have to concatenate a list of the coordinates to pass to the drawing functions using the `function ! list` form of invocation.

Table 56 provides a list of the basic graphics functions Remember, the window parameter is optional: it defaults to the value of keyword `&window`. The basic attributes used in line drawing are shown in Table 57..

Table 56 Basic line-drawing functions.

`Clip(W,x,y,w,h)`	Set a clipping rectangle with a corner at position (x,y), and width w and height h. The default w and h values are to the right and bottom edges of the window. Subsequent drawing outside the bounds will have no effect. When called without the (x,y) parameters, clipping is turned off and the entire canvas can be written. Returns the window, W.

`DrawArc(W,x,y,w,h)`
`DrawArc(`
`  W,x,y,w,h,theta,phi)`
	Draws an ellipse in the rectangle of width w and height h and a corner at position (x,y). If theta and phi are given, it draws an arc of the ellipse starting at angle theta and extending for angle phi.
	
	Returns the window, W.

`DrawCircle(W,x,y,r)`
`DrawCircle(`
`  W,x,y,r,theta,phi)`
	Draws a circle with center at position (x,y) and radius r. If theta and phi are given, it draws an arc of a circle starting at angle theta and extending for angle phi.
	
	Returns the window, W.

`DrawCurve(` ` W,x1,y1,x2,y2,...,xn,yn)`	Draws a smoothed curve from (x1,y1) to (xn,yn) passing through the intermediate points in order. If xn=x1 and yn=y1, then the curve will be closed and smoothed through point (x1,y1) as well. Returns the window, W.
`DrawLine(` ` W,x1,y1,x2,y2,...,xn,yn)`	Draws a connected sequence of straight lines. A straight line is drawn from point (x1,y1) to (x2,y2), then a line from (x2,y2) to (x3,y3), and so on to (xn,yn). Returns the window, W.

Table 56 Basic line-drawing functions.

`DrawPoint( W,x1,y1,x2,y2,...,xn,yn)`	Draws a point at each position (x,y) given. Returns the window, W.
`DrawPolygon( W,x1,y1,x2,y2,...,xn,yn)`	Equivalent to DrawLine with a final line segment back to (x1,y1). Returns the window, W.
`DrawRectangle( W,x1,y1,x2,y2)`	Draws the rectangle with opposite corners (x,y) and (x+w,y+h). Parameters w and h default to the right and bottom edges of the window. Returns the window, W.
`DrawSegment( W,x1,y1,x2,y2,...,xn,yn)`	Similar to DrawLine, except it draws disconnected line segments between pairs of points: (x1,y1) to (x2,y2), then (x3,y3) to (x4,y4), etc. Returns the window, W.
`EraseArea(W,x,y,w,h)`	Fills the rectangle with opposite corners (x,y) and (x+w,y+h) with the background color. Parameters w and h default to the right and bottom edges of the window. Returns the window, W.

Table 57 Attributes for line drawing.

`"dx=`*num*`"`	integer to be added to each x parameter passed to a graphics function to determine the actual pixel position in the window.
`"dy=`*num*`"`	integer to be added to each y parameter passed to a graphics function to determine the actual pixel position in the window.

Table 57 Attributes for line drawing.

`"linestyle=`*s*`"`	string describing the style of the lines to be drawn. The options differ based on the underlying graphics system. For X-windows, the options are `solid`, `dashed`, and `striped`. The `dashed` lines have gaps. The `striped` lines are `dashed` with their gaps filled with the background color. In the beta version of Icon for Windows, the options are `dashed`, `dotted`, `solid`, `dashdotted`, and `dashdotdotted`. Style `striped` is treated as `dashed`. The default style is `solid`.
`"linewidth=`*num*`"`	number specifying the width of the lines drawn.

17.3.2 Examples of line drawings

Example. The differences in some of the line-drawing functions are shown in Figure 38 on page 157. The code to draw the lines is shown in Figure 37 on page 156. Notice that DrawCurve is much like DrawLine, except that the lines are curved and the junctions between the line segments are smoothed. When the initial point is equal to the last point in DrawCurve, the curve is also smoothed through that point.

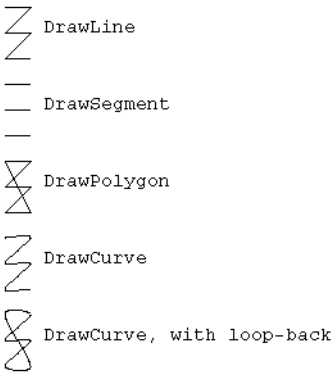
Figure 37 Drawing lines.

```
link graphics
procedure main()
WOpen("size=400,350") | stop("can't open win-
dow")
L:=[]
every i:=10 to 50 by 20 & j:=10 to 30 by 20 do {
     put(L,j,i)
}
DrawLine!L
     DrawString(L[7]+10,L[8],"DrawLine")
every L[2 to *L by 2] +:=60
DrawSegment!L
     DrawString(L[7]+10,L[8],"DrawSegment")
every L[2 to *L by 2] +:=60
DrawPolygon!L
     DrawString(L[7]+10,L[8],"DrawPolygon")
every L[2 to *L by 2] +:=60
DrawCurve!L
     DrawString(L[7]+10,L[8],"DrawCurve")
every L[2 to *L by 2] +:=60
DrawCurve!(L ||| [L[1],L[2]])
     DrawString(L[7]+10,L[8],"DrawCurve, with
```



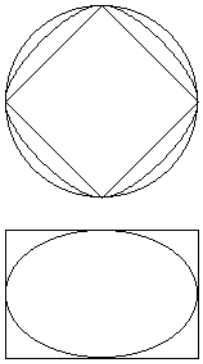
```
loop-back")

WriteImage("lines.gif")
WDone()
end
```

Figure 38 *Lines drawn by Figure 37*



Example. Figure 39 on page 157 shows several closed figures. The diamond was drawn with a DrawPolygon. In the top figure, the outer curve was drawn by DrawCircle. The curve between them was drawn by DrawCurve. The bottom figure is the result of DrawRectangle and DrawArc with the same parameters.

Figure 39 *Closed figures.*



Example. The spiral in Figure 41 on page 158 was drawn by the code in Figure 40 on page 157. It is done by drawing arcs of concentric circles while varying the angles theta and phi.

Figure 40 *Draw a spiral.*

```
link graphics
procedure main()
   local r
```



```
      WOpen("size=400,400") | stop("can't open window")
      WAttrib("dx=200","dy=200")
      every r := 5 to 180 by 2 do
        DrawCircle(0,0,r,dtor(r),2*dtor(r))
      WriteImage(&window,"spiral.gif",-200,
        -200,400,400)
      WDone()
end
```

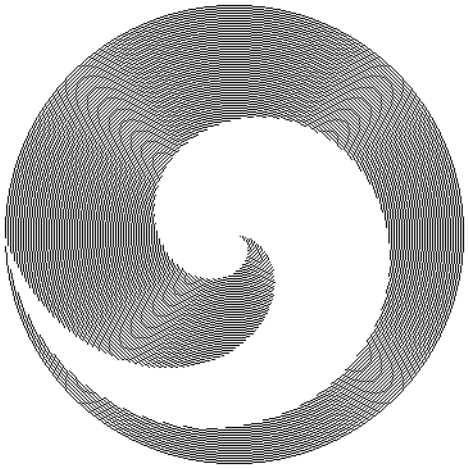


Figure 41 Spiral—the effects of angles.

Example. Figure 42 on page 158 explores more precisely what the two angles mean in DrawCircle and DrawArc. The top figure explores circles and the bottom, ellipses drawn with DrawArc.Lines at angles theta and theta+phi are drawn from the center of both figures. The top shows a circle and an arc of a slightly smaller concentric circle. The angles clearly mark the beginning and ending positions of the arc. For the bottom figure, an arc is drawn in a rectangle, but the angles from the center of the rectangle do not correspond to the beginning and ending of the arc.

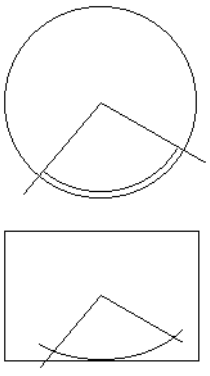


Figure 42 Angles in DrawCircle and DrawArc.

17.4 Filled areas

17.4.1 Basic area-filling functions.

You can draw several kinds of areas and have them filled-in with the background color or a pattern. The relevant functions are given in Table 58. Several examples of filled figures are shown in Figure 43

Table 58 Functions for filling areas.

`EraseArea(W,x,y,w,h)`	Fills the rectangle with opposite corners (x,y) and (x+w,y+h) with the background color. Parameters w and h default to the right and bottom edges of the window.
`FillArc(W,x,y,w,h)`	fills an ellipse within the rectangular area which has one corner at position (x,y) and width w and height h.
`FillArc( W,x,y,w,h,theta,phi)`	fills a wedge of an ellipse within the rectangular area which has one corner at position (x,y) and width w and height h. The wedge subtends the arc that would be drawn by DrawArc(`W,x,y,w,h,theta, phi)`.
`FillCircle(W,x,y,r)`	fills a circle with center (x,y) and radius r.
`FillCircle( W,x,y,r,theta,phi)`	fills a wedge of a circle with center (x,y) and radius r. The wedge starts at angle theta and extends for angle phi.
`FillPolygon( W,x1,y1,x2,y2,...,xn,yn)`	fills the contained areas in the polygon that would be drawn by `DrawPolygon( W,x1,y1, x2,y2, ...,xn,yn)`
`FillRectangle(W,x,y,w,h)`	fills the rectangle with a corner at (x,y) and width w and height h.
`Pattern(W,s)`	establishes the pattern to be used for `Fill...` function calls. The pattern specification s may be one of the predefined names shown in Figure 44 on page 161, or it may be a bi-level image as described in Section 17.4.4 on page 161.

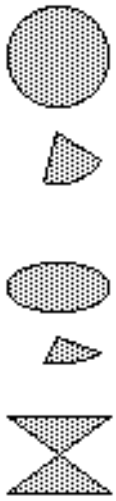


Figure 43 Some filled figures

17.4.2 Fill style

You choose what style fill to use with the window attribute `fillstyle` :

- `solid`—the area is filled with the foreground color.
- `textured`—the area is filled with the fill pattern. All pixels in the area are overwritten with pixels from the pattern.
- `masked`—the foreground colors in the pattern overwrite the pixels in the area. The background color in the pattern leaves the pixels in the area unchanged.

So if you are going to use patterns, you must set the `fillstyle` attribute to something other than `solid`.

17.4.3 Patterns

The actual pattern used is specified by window attribute `pattern`. You will typically set the attribute by calling the procedure

```
Pattern(W,spec)
```

The pattern specification can be one of the predefined pattern names shown in Figure 44 or it can be a bi-level image.

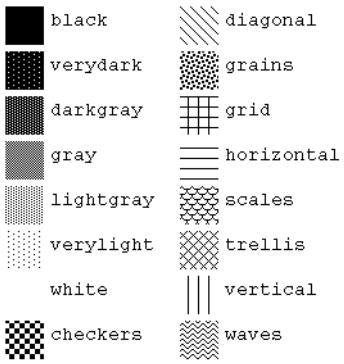


Figure 44 Fill patterns.

17.4.4 Bi-level images

A bi-level image is a rectangular bit pattern where 1 specifies the foreground color and 0 specifies the background color. A bi-level image is itself specified by a string of the form:

"*width,#hexdata*"

The `width` is an integer specifying how wide the pattern is, in pixels. The `hexdata` is comprised of hexadecimal digits specifying four bits apiece. The four bits specify up to four horizontally contiguous pixels, the least-significant bit specifying the leftmost.

While the number of pixels horizontally is specified by width, the number vertically is deduced from the number of hexadecimal digits provided. The pattern is filled in by taking the hexadecimal digits left to right to fill a row, then if there are any more hexadecimal digits, starting the next row of pixels with the next hex digit. If width is not evenly divisible by four, some bits specified by the last hex digit in a row will be ignored.

When used as a fill, the bi-level image is tiled to fill the area required. If you are just going to fill a rectangular area exactly the size of the bi-level image, you can use the function DrawImage:

```
DrawImage(W,x,y,s)
```

which will place the bi-level image on the screen with its upper left corner at position (`x`,`y`). When `s` has the form

"*width,#hexdata*"

it is written as `textured`: the 1 bits are written as the foreground color and the 0 bits are written as the background color. If `s` has the form

"*width,~hexdata*"

(*i.e.* ~ is substituted for #, but otherwise the specification is identical) it is written as `masked`: the 1's are written as foreground color but the 0's are not written, leaving those pixels unmodified.

A full description of DrawImage is given in Table 67 on page 174.

17.4.5 Fill attributes

Window attributes related to filling areas are shown in Table 59

Table 59 Fill-related attributes.

`"bg=s"`	string s specifies the background color.
`"drawop=s"`	string s specifies either "copy" or "reverse". With "copy" the new pixels replace the existing pixels. With "reverse", the new pixels will combine with the existing pixels in an exclusive-or fashion converting foreground-colored pixels into background-colored and vice versa.
`"fg=s"`	string s specifies the foreground color.
`"fillstyle=s"`	string s specifies whether a pattern is to be used when drawing filled figures, and if so, how. The options are `"solid"`, `"textured"`, and `"masked"`. If `"solid"` is specified, figures are filled with the foreground color. If `"textured"` is specified, both the foreground and background colors of the pattern are used. If `"masked"`, only the foreground colors of the pattern are written; pixels corresponding to the background colors are left unchanged. The default is `"solid"`.
`"pattern=s"`	string s specifies the fill pattern to be used, either a pattern name (Figure 44) or a bi-level image (section 17.4.4).
`"reverse=s"`	string `s` is either `"on"` or `"off"`. When changed from `"on"` to `"off"` or `"off"` to `"on"`, the values of `bg` and `fg` are swapped.

17.4.6 Example of filled areas

Figure 45 shows a collection of beveled figures created by writing filled rectangles and circles. The code producing this figure is shown in Figure 46. The trick is to write lighter and darker copies of the figures offset slightly up to the left and down to the right, and then the figure itself in the background color in the center.

Figure 46 Code for beveled figures.

Figure 45 Beveled figures.

```
link graphics
procedure main()
   local r,flip
   WOpen("size=225,400","bg=very light gray") |
     stop("can't open window")

   bevrectangle(10,10,100,100,-1)
   bevrectangle(35,35,50,50,1)

   bevcircle(160,160,50,-1)
   bevcircle(160,160,25,1)

   flip:=-1
   every r:=90 to 5 by -5 do {
     bevcircle(100,300,r,flip)
     flip:=-flip
   }
   WriteImage("bevfig.gif")
   WDone()
end

procedure bevrectangle(x,y,w,h,d)
   local oldfg,nw,se
   oldfg:=Fg()
   nw:="white"
   se:="grey"
   if d<0 then {nw:=:se; d:=-d}
   w-:=d
   h-:=d
   Fg(nw)
   FillRectangle(x,y,w,h)
   Fg(se)
   x+:=d
```



```
   y+:=d
   FillRectangle(x,y,w,h)
   Fg(Bg())
   FillRectangle(x,y,w-d,h-d)
   Fg(oldfg)
return
end

procedure bevcircle(x,y,r,d)
   local oldfg,nw,se
   oldfg:=Fg()
   r:=r-d
   nw:="white"
   se:="grey"
   if d<0 then {nw:=:se; d:=-d}
   Fg(nw)
   FillCircle(x-d,y-d,r)
   Fg(se)
   FillCircle(x+d,y+d,r)
   Fg(Bg())
   FillCircle(x,y,r)
   Fg(oldfg)
return
end
```

17.5 Text

You can treat a window as a video terminal using functions `WWrite()`, `WWrites()`, `WRead()`, and `WReads()` for `write()`, `writes()`, `read()`, and `reads()`. The text-related functions are shown in Table 60.

Table 60 Text-related window functions.

`DrawString(W,x,y,s)`	writes the string `s` starting at position (`x`,`y`) in window `W` without modifying the text cursor.
`Font(W,s)`	sets the font in window `W` to that specified by string `s`, or fails if it cannot be done.
`GotoRC(W,r,c)`	sets the text cursor in window `W` to row `r` and column `c`. `GotoRC(W)` sets the row and column to 1,1.
`GotoXY(W,x,y)`	sets the text cursor position in window `W` to position (`x`,`y`). `GotoXY(W)` sets the position to (0,0).
`TextWidth(W,s)`	returns the number of pixels of width that string `s` would require if written in window `W`.

Table 60 Text-related window functions.

`WRead(W)`	reads a line typed into window `W` in the manner of `read`. Displays the text cursor and echoes the characters typed if window attributes `cursor` and `echo` allow it.
`WReads(W,i)`	reads `i` characters (default one) typed into window `W` in the manner of `reads`. Displays the text cursor and echoes the characters typed if window attributes `cursor` and `echo` allow it.
`WWrite( W,s1,s2,...,sn)`	writes the strings `s1`, `s2`, .., `sn` in window `W` in the manner of `write`—followed by moving the cursor to the beginning of the next line, scrolling if required.
`WWrites( W,s1,s2,...,sn)`	writes the strings `s1`, `s2`, .., `sn` in window `W` in the manner of `writes`—scrolling if required by any `\n` characters written.

A window is by default drawn large enough for twelve lines of eighty columns of text.

There is a text cursor that starts at the upper left corner of the window. The cursor is moved as text is written or read, and if it would go off the bottom, the window scrolls.

The text cursor is not visible by default. You must set the window attribute `cursor` to `on` to see the cursor (and set it back to `off` to make the cursor invisible), see Table 61. E.g.

```
WAttrib(W,"cursor=on")
WAttrib(W,"cursor=off")
```

The cursor is the underscore character, does not blink, and is only displayed if attribute `cursor` is set `on` and Icon is waiting input `WRead` or `WReads`. (Good luck finding it.)

Table 61 Window attributes related to text.

`"ascent"`	the number of pixels the current font extends above the base line. Read only.
`"descent"`	the number of pixels the current font extends below the base line. Read only.
`"col=num"`	column of the text cursor (in characters).
`"cursor=`*`s`*`"`	controls the visibility of the text cursor in the screen: `"on"` or `"off"`, `"off"` by default.

Table 61 Window attributes related to text.

`"echo=`*`s`*`"`	flag controlling whether characters typed to `WRead` and `WReads` are displayed on the screen: `"on"` or `"off"`, `"on"` by default.
`"font=`*`s`*`"`	current text font.
`"fheight"`	height of the current text font, top to bottom. Read only.
`"fwidth"`	width of the current text font. Read only.
`"leading=`*`num`*`"`	the number of pixels between successive lines of text.
`"row=`*`num`*`"`	row of the text cursor (vertical position in characters).
`"x=`*`num`*`"`	horizontal position of the text cursor (in pixels).
`"y=`*`num`*`"`	vertical position of the text cursor (in pixels).

The cursor may be moved explicitly to a screen position with

```
GotoRC(W,row,col)
```

where the optional parameter `W` specifies the window, `row` specifies the row, and `col` specifies the column, or

```
GotoXY(W,x,y)
```

where the optional parameter `W` specifies the window, and `x` and `y` specify the pixel position. Notice that `row` and `col` specify the distance down and across, while `x` and `y` specify the position across and down.

Rather than writing

```
GotoXY(W,x,y)
WWrites(W,text)
```

you can use

```
DrawString(W,x,y,text)
```

which will not change the cursor position. `DrawString` is probably more appropriate for writing captions on graphics.

When you are reading text in a window, the characters the user types will only be displayed on the screen if window attribute `echo` is set to `on`. If `echo` is `off`, the characters will not be written. By default, characters will be echoed.

You can select the font that will be displayed in a window. You can change the font by calling function

```
Font(s)
```


which will set the font to that specified by the string s.

Fonts are specified by strings of the form

family,style,size

where *family* gives the family name of the font, *style* gives such characteristics as *bold*, *italic*, or *bold,italic*, and *size* specifies the size of the font. The style and size parts of the specification are optional.

There are four built-in font families, shown in Table 62. These, in four styles and several sizes, are shown in Figure 47, along with a the non-portable family *Times*.

Table 62 Built-in font families

	monospaced	proportionally spaced
sans-serif	`mono`	`sans`
serif	`typewriter`	`serif`

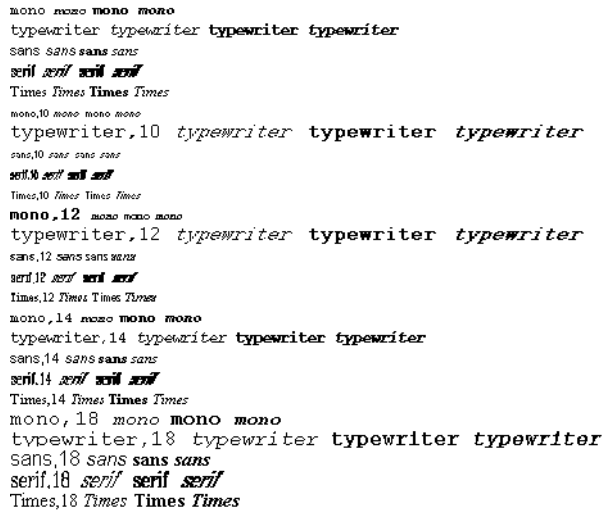


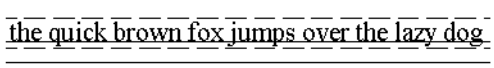
Figure 47 Fonts.

Windows have a number of attributes related to the currently active font, see Table 61. Attributes `fheight` and `fwidth` give the maximum height and width of the current font. The a proportionally-spaced font, some characters may require less space.

Characters extend up- and down-wards from a line.Attribute `ascent` gives the number of pixels upwards the font extends; `descent` the number of pixels downwards. Attribute `leading` specifies the distance between successive lines of text. Only the `leading` attribute can be assigned new values; the others are fixed for a font. Figure 48 shows some of the font attributes. The two sol-

id lines are the base-lines of text `leading` pixels apart. The two dashed lines are drawn at the first baseline minus `ascent` and plus `descent` pixels.

Figure 48 Font attributes.



Example. The code in Figure 49 displays a moving sign. The trick is to repeatedly write the message over one pixel to the left each time. The message text must end in a blank to clean out the rightmost pixels. A certain amount of complexity in the code is the result of GotoXY refusing to set the cursor position outside the window: we established a clipping region within the window and shifted the message one character at a time.

Figure 49 Moving sign.

```
link graphics
procedure main()
   local msg,fontwidth
   WOpen("size=300,40") |
      stop("can't open window")
   Font("TimesRoman,bold,24") |
      stop("can't set font")
   fontwidth:=WAttrib("fwidth")
   Clip(fontwidth,1,
     WAttrib("width")-2*fontwidth,
     WAttrib("height"))
   repeat {
    msg := "the quick brown fox jumps_
           over the lazy dog "
    msg := repl(" ",
          WAttrib("width")/TextWidth(" ")+2)|| msg
    while *msg>0 do {
     every GotoXY(
          fontwidth-(1 to TextWidth(msg[1])),
           30) do {
          WWrites(msg)
     }
     msg:=msg[2:0]
    }
   }
end
```

17.6 Colors

17.6.1 Color specifications and names

Colors are specified in the red-green-blue system by giving the intensities of each of these components. Each intensity is specified in 16 bits: The minimum

intensity is zero; the maximum is 65535. The colors are specified by strings giving either a decimal or a hexadecimal specification of the intensities red, green, and blue in one of the forms:

```
"R,G,B"
"#rgb"
"#rrggbb"
"#rrrgggbbb"
"#rrrggggbbbb"
```

where R, G, and B are decimal numbers in the range 0-65535 and r, g, and b represent hexadecimal digits.

Rather than using the numeric color specification, you can specify colors using names of the form:

lightness saturation modifier hue

There are fourteen built-in hues that can be referred to by name; they are shown in Table 63.

Table 63 The built-in hues.

hue	*decimal specification*
black	0,0,0
blue	0,0,65535
brown	32767,16383,0
cyan	0,65535,65535
grey	32767,32767,32767
green	0,65535,0
magenta	65535,0,65535
orange	65535,16383,0
pink	65535,32767,40959
purple	32767,0,65535
red	65535,0,0
violet	49151,32767,65535
white	65535,65535,65535
yellow	65535,65535,0

The grammar in Figure 50 shows the full form of a color name. Only the hue is required.The modifier is either a *hue* or a *hue* with the suffix *-ish* written in cor-

rect English (e.g. *red* becomes *reddish*). If the color specifies two hues, the actual hue is half way between them. If it specifies a *huish hue*, then the actual hue is three-quarters of the way from the first to the second. The saturation `pale` is a synonym for `very light`, and `deep`, for `very dark`. The elements of a color name are separated by either blanks or hyphens (`separator`).

Figure 50 *Grammar for color names.*

```
colorName = lightness_opt saturation_opt
       modifier_opt hue.
lightness_opt = lightness separator | .
lightness = very separator light | pale
     | light
     | medium
     | dark
     | very separator dark | deep
     .
saturation_opt = saturation separator | .
saturation = weak
     | moderate
     | strong
     | vivid
     .
modifier_opt = modifier separator | .
modifier = hue | huish .
hue  = black| blue| brown
     | cyan| gray| green
     | magenta| orange| pink
     | purple| red| violet
     | white| yellow
     .
huish= blackish| bluish| brownish
     | cyanish| grayish| greenish
     | magentaish| orangish| pinkish
     | purplish| reddish| violetish
     | whitish| yellowish
     .
separator = " " | "-" .
```

An example of how lightness and saturation modify a color is shown in Table 64.

Table 64 Lightness and saturation.

color name	specification
`very light weak red`	`57343,51881,51881`
`very light moderate red`	`60073,49151,49151`
`very light strong red`	`62804,46420,46420`

Table 64 Lightness and saturation.

color name	specification
`very light vivid red`	`65535,43689,43689`
`light weak red`	`49151,38228,38228`
`light moderate red`	`54612,32767,32767`
`light strong red`	`60073,27306,27306`
`light vivid red`	`65535,21845,21845`
`medium weak red`	`40959,24575,24575`
`medium moderate red`	`49151,16383,16383`
`medium strong red`	`57343,8191,8191`
`medium vivid red`	`65535,0,0`
`dark weak red`	`27306,16383,16383`
`dark moderate red`	`32767,10922,10922`
`dark strong red`	`38228,5461,5461`
`dark vivid red`	`43690,0,0`
`very dark weak red`	`13653,8191,8191`
`very dark moderate red`	`16383,5461,5461`
`very dark strong red`	`19114,2730,2730`
`very dark vivid red`	`21845,0,0`

17.6.2 Color correction

As explained in Foley, *et al.*[1], CRTs and film are non-linear, and the intensity, I, is related to the level applied to the pixel, V, by the formula

$$I = K \cdot V^{\gamma}$$

where K and γ (gamma) are constants.

Icon allows you to select a gamma correction to be used when Icon passes color specifications to the underlying graphics system, with 1.0 giving no color correction and larger values giving less saturated, lighter colors.

1. Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Huges, *Computer Graphics: Principles and Practice, Second Edition,* Addison Wesley, 1990, p565.

17.6.3 Palettes, images

Palettes allow you to refer to colors or shades of gray with single characters. The size of &cset, 256 in current Icon implementations, limits the number of colors or graytones possible. You can use palettes instead of bit maps in the DrawImage function to draw rectangular, colored or grayscale images in the window.

Palettes are predefined; you don't get to create your own. The grayscale palettes are named `"g2"`, `"g3"`, `"g4"`, ..., `"g64"`. The color palettes are named `"c1"`, `"c2"`, `"c3"`, `"c4"`, `"c5"`, and `"c6"`.

The function DrawImage (see 17.4.4 on page 161) can draw a color or grayscale image as follows:

```
DrawImage(W,x,y,"width,palette,chars")
```

will fill the rectangular area in window *W* with a corner at position (*x*,*y*) and width *width.* The colors specified associated with the characters *chars* in palette *palette* are used to fill in the pixels from left to right, top to bottom.

The characters in gray palette `g`*i* are the first *i* characters in the string[2]

```
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_
abcdefghijklmnopqrstuvwxyz{}"
```

The first character is black, the last character is white, and the characters in between represent uniformly spaced shades of gray.

The color palette c1 is rather arbitrary in its design. It is shown in Table 65 which is based on a table in Appendix F of Icon Project manual IPD255.[3] The table was created by constructing color names from the column headings concatenated with the row headings and asking function PaletteColor what character is closest to that color in the c1 palette. The gray areas were omitted from the table in IPD255 as not being part of the designed palette.

Table 65 Color palette `c1`.

	very dark	dark	medium	light	very light	weak
black	0	0	0	2	4	0
gray	1	2	3	4	5	3
white	2	4	6	6	6	6
brown	!	p	?	C	9	2

2. Note we are using Icon continuation conventions: an underscore as the last character of a line continues a string with the first nonwhitespace character of the next line. The underscore is not part of the string.

3. Clinton L. Jeffery, Gregg M. Townsend, Ralph E. Griswold, Graphics Facilities for the Icon Programming Language, Version 9.0, Icon Project, University of Arizona, IPD255, July, 1994.

Table 65 Color palette `c1`.

red	n	N	A	a	#	@
orange	o	O	B	b	$	%
red-yellow	p	P	C	c	&	|
yellow	q	Q	D	d	,	.
yellow-green	r	R	E	e	;	:
green	s	S	F	f	+	-
green-cyan	t	T	G	g	*	/
cyan	u	U	H	h	‘	'
cyan-blue	v	V	I	i	<	>
blue	w	W	J	j	(	)
blue-magenta	x	X	K	k	[	]
purple	x	X	K	k	[	]
magenta	y	Y	L	l	{	}
magenta-red	z	Z	M	m	^	=
violet	X	K	8	[	6	4
pink	N	A	7	^	6	4

The color palettes `"c2"` through `"c6" are the` *uniform color palates.* The basic idea is that palette `c`*n* gives you n levels of intensity of each of the colors red, green, and blue. You can form n^3 colors by combining intensities of their red, green, and blue components.

The first n^3 characters of palette `c`*n* select the colors. The intensity levels in the `c`*n* palette are numbered from 0 through n-1. If P is the string of characters in the palette and r, g, and b are the intensity levels for red, green, and blue, then the color with intensities r,g,b is selected by character

```
P[r*n*n+g*n+b+1]
```

Every color with equal intensities of red, green, and blue is a shade of gray, but they are too few to give good rendering of grayscale images, so additional shades of gray were added. The characters for these additional grays are tacked on to the end of the palette.

The `c`*n* palette adds an additional n-1 shades of gray between each successive

rgb gray level. That gives n^2-2n+1 additional characters attached to the end of the palette representing the additional levels of gray and a total of n^2-n+1 gray levels in total.

Table 66 shows the characters in the color palettes.

Table 66 Color palettes

Palette	Color Characters	Additional grays
`c1`	(see Table 65)	`"0123456"`
`c2`	`"kbgcrmyw"`	`"x"`
`c3`	`"@ABC...XYZ"`	`"abcd"`
`c4`	`"012...89ABC... YZabc...yz{}"`	`"$%&*+-/?@"`
`c5`	`"\x00\x01\x02...yz{|"`	`"}~\d\x80... \x8a\x8b\x8c`
`c6`	`"\x00\x01\x02... \xd6\xd7"`	`"\xd8\xd9\xda... \xee\xef\xf0"`

There are four functions to use with palettes: PaletteColors, PaletteChars, PaletteGrays, and PaletteKey. Their meanings are shown in Table 67.

Table 67 Palette functions.

`DrawImage(W,x,y,s)`	will fill the rectangular area in window `W` with an upper left corner at position (`x`,`y`) with the image specified by string `s`. String *s* has one of three forms: *`"width,#hexdata"`* "*width*`,~`*hexdata*" *`"width,palette,chars"`* In all cases, the width *`width`* specifies the width of the rectangular area to fill. The height is determined by the number of characters or bits to write.
`DrawImage(W,x,y,` *`"width,#hexdata"`*`)`	the hexdata is a bi-level image.It is written as `textured`: the 1 bits are written as the foreground color and the 0 bits are written as the background color.

Table 67 Palette functions.

`DrawImage(W,x,y, "`*`width,~hexdata`*`")`	the hexdata is a bi-level image. It is written as `masked`: the 1's are written as foreground color but the 0's are not written, leaving those pixels unmodified.
`DrawImage(W,x,y, "width,palette,chars")`	the colors specified associated with the characters *`chars`* in palette *`palette`* are used to fill in the pixels from left to right, top to bottom. Character \377 and character ~ (if it is not contained in the palette) specify transparent pixels which will not overwrite the previous value. Commas and spaces, if the palette does not contain them, can be inserted to ease readability; they will not display.
`PaletteChars(palette)`	returns as a character string the characters in the palette. For a grayscale palette, the characters will represent the intensities of gray from black to white in order. For uniform color palettes, cn, the first n^3 characters can be indexed by n intensity levels (0, 1, ..., n-1) for red, green, and blue, to select the character representing that combination of intensities, `P[r*n*n+g*n+b+1]`. The last n^2-2n+1 characters represent the additional gray levels in order from darkest to lightest. Since the trailing grays have gaps (the grays in the regular palette), if you intend to index into the grays, use function `PaletteGrays` to get a complete list.
`PaletteColor( palette,s)`	returns the color in palette represented by the single character s.
`PaletteGrays(palette)`	returns a string containing in order from black to white the characters from the palette representing levels of gray.
`PaletteKey(palette,s)`	returns a character from the palette which is close to the color specified by string s.

Table 67 Palette functions.

`ReadImage(W,s1,x,y,s2)`	will read the image in the file named `s1` into the window `W` with its upper-left corner at position (`x`,`y`). If `s2` is specified, the colors in the image are converted into colors in palette `s2`.

17.6.4 Mutable colors

Icon gives you access to your system's color map, if it is changeable, via *mutable colors.* You can allocate a mutable color and change the color it represents repeatedly. Whenever you assign a mutable color a new actual color, all the pixels written with that mutable color change to the new color. (The internal representation of a mutable color is called a color map entry.)

You allocate a mutable color assigning it the initial color s with function `NewColor(s)`. `NewColor` returns a negative integer to represent the mutable color. You can assign it a different color with the function `Color(n,s)`, where `n` is the mutable color and `s` is a color specification. On some systems, when a color map entry is no longer needed, it can be returned to a pool for subsequent reallocation with the function `FreeColor(n)`.

Table 68 Mutable color functions.

`NewColor(W,s)`	allocates a changeable color map entry for a new mutable color and assigns it initially the color specified by s, and returns a small negative integer to represent the mutable color. Fails if no changeable color map entry is available.
`Color(` `  W,n1,s1,n2,s2,...)`	sets each mutable color n_i to the corresponding color specified by s_i.
`FreeColor(n1,...)`	frees the color map entries for all n_i. Not available in all implementations. Bad things may happen if any pixels are still assigned the color being freed.

17.7 Pixel rectangles, moving, saving, restoring

There are several functions that operate on rectangular areas of pixels. You can copy such an area, write it to a file, or examine the pixels in

it. These functions are given in Table 69

Table 69 Pixel rectangle functions.

`CopyArea( W1,W2,x1,y1,w,h, x2,y2)`	copy a rectangular area of width `w` and height `h` from position (`x1`,`y1`) in `W1` to position (`x2`,`y2`) in window `W2`. The (`x`,`y`) values specify the upper-left corner. If `W2` is omitted, `W1` is both the source and destination window. If `W1` and `W2` are both omitted, `&window` is the source and destination.
`Pixel(W,x,y,w,h)`	generates the colors of the pixels in the rectangle of width `w` and height `h` with its upper left corner in position (`x`,`y`) of window `W`. The pixels are generated by rows, left to right, top to bottom.
`ReadImage(W,s1,x,y)` `ReadImage( W,s1,x,y,s2)`	will read the image in the file named `s1` into the window `W` with its upper-left corner at position (`x`,`y`). If `s2` is specified, the colors in the image are converted into colors in palette `s2`.
`WriteImage( W,s,x,y,w,h)`	writes the rectangle of pixels of width `w` and height `h` with its upper left corner in position (`x`,`y`) of window `W` into the file named `s`. Normally Icon writes the file in GIF format, but it may allow other extensions on the file name to choose other formats.

17.8 Events

Mouse operations in a window or characters typed in a window when no WRead or WReads is pending cause *events* to be queued. You can process an event by calling the function `Event(`*`window`*`)` which will return the *event code* and will set the values of certain keywords.

`Event()` will wait for an event to occur. If you don't want to wait, you can execute `*Pending(`*`window`*`)` which will tell you the length of the event queue for the window. If it is zero, there are no events pending. However, each event is currently represented by three elements in the list, so the length of the pending list is three times the number of events queued up.

The event codes returned for keyboard characters are single character strings. The events returned for the mouse buttons and for some other keyboard keys are integers. The integers corresponding to certain events are the values of some keywords. For other special keys, there is a file of definitions, `keysyms.icn`, that you can include to refer to the key by name rather than number:

```
$include "keysyms.icn"
```


There are three keyboard keys that do not themselves cause events but rather modify other events: SHIFT, CONTROL, and META (or ALT). The states of those keys when another event occurs are preserved in the keywords `&shift`, `&control`, and `&meta`.

Figure 51 is a program that will display in a window and write to a file the events that occur in the window. The call to Event returns the event code and

Figure 51 Show events.

```
link graphics
procedure main()
   WOpen("size=600,400") | stop("can't open window")
   f:=open("tstev1.txt","w")
   repeat {
     e:=Event()
     WWrite(e,",",keys(),",",
          &x,",",&y,",",&interval)
     write(f,e,",",keys(),",",
          &x,",",&y,",",&interval)
     if e===("q"|"Q") then break
   }
   close(f)
   WClose()
end
procedure keys()
return((&shift & "1")| "0") ||
     ((&meta & "1")| "0") ||
     ((&control & "1")| "0")
end
```

sets the values for some keywords. Keywords `&x` and `&y` give the position in the window of the cursor when the event occurred. Keyword `&interval`, on some systems, gives the time in milliseconds since the previous event occurred. The keyword `&shift` succeeds yielding &null if the SHIFT key was pressed during the event, but fails if it wasn't. Similarly, `&meta` and `&control` report the META (ALT) and CONTROL keys.

Figure 52 shows a program to draw circles in a window under mouse control. The repeat loop is an example of event-driven programming. A "Q" typed on the keyboard will break out of the loop. Pressing the left button of the mouse saves the cursor's x and y position as the center of the circle. Holding down the button and dragging the mouse draws a circle out to the current mouse position. Releasing the button leaves the circle drawn. As the mouse is moved with the left button held, the circle is shown. The trick for showing the circle uses the "drawop=reverse" attribute assignment. A circle is drawn. Then as the mouse moves, the previous circle is redrawn, which erases it, and the new circle is drawn. As each circle is finished, it's center is marked with crossed lines and

the center of the previous circle is unmarked.

Figure 52 Code to draw circles.

```
link graphics
procedure main()
   local radius,x,y,px,py
   px:=py:=-10
   WOpen("size=400,300","drawop=reverse") |
     stop("can't open window")
   repeat { case Event() of {
     "q"|"Q": break
     &lpress: DrawCircle(x:=&x, y:=&y, radius := 0)
     &ldrag:  {
          DrawCircle(x,y,radius)
          DrawCircle(x, y, radius:=sqrt((&x-x)^2+(&y-y)^2) )
          }
     &lrelease: {
          DrawSegment(px-5,py,px+5,py,px,py-5,px,py+5)
          DrawSegment(x-5,y,x+5,y,x,y-5,x,y+5)
          px:=x; py:=y
          }
     }
   }
   DrawSegment(px-5,py,px+5,py,px,py-5,px,py+5)
   WriteImage("circles.bmp")
end
```

The keywords and functions used by the event system are shown in Table 70.

Table 70 Event keywords and functions.

`Active()`	returns a window that has one or more events pending. It will wait until such a window is available. It will poll the open windows in a different order each time to avoid starvation. It fails if there are no open windows.
`&col`	The text column of the event reported by the most recent call of Event.
`&control`	succeeds returning &null if the CONTROL key was pressed during the event reported by the most recent call of Event, otherwise fails.

Table 70 Event keywords and functions.

`Enqueue(W,e,x,y,cms,i)`	adds an event to the end of the list of pending events in window W. The event code is e. The position is (x,y). The string cms indicates the state of the control, meta, and shift keys: if the character "c" is in the string, the event reports the CONTROL key was pressed, and similarly "m" and "s" for META (ALT) and SHIFT. Parameter i gives the time in milliseconds for &interval. By default, e is &null, x is 0, y is 0, cms is "", and i is 0.
`Event(W)`	waits for an event in window W (if necessary), removes the first event from W's event queue, sets &control, &shift, &meta, &interval, &row, &col, &x, and &y, and returns the event code.
`&interval`	the interval, in milliseconds, between the event reported by the most recent call of Event and the event preceding it. This is not meaningful on all systems.
`&ldrag`	the integer event code returned when the mouse is moved while the left mouse button is pressed.
`&lpress`	the integer event code returned when the left mouse button is pressed.
`&lrelease`	the integer event code returned when the left mouse button is released.
`&mdrag`	the integer event code returned when the mouse is moved while the middle mouse button is pressed.
`&meta`	succeeds returning &null if the META (ALT) key was pressed during the event reported by the most recent call of Event, otherwise fails.
`&mpress`	the integer event code returned when the middle mouse button is pressed.
`&mrelease`	the integer event code returned when the middle mouse button is released.
`Pending(W)`	returns the list of events pending in window W. The list is empty is no events are pending.

Table 70 Event keywords and functions.

`&rdrag`	the integer event code returned when the mouse is moved while the right mouse button is pressed.
`&resize`	the integer event code returned when the window is resized.
`&row`	The text row of the event reported by the most recent call of Event.
`&rpress`	the integer event code returned when the right mouse button is pressed.
`&rrelease`	the integer event code returned when the right mouse button is released.
`&shift`	succeeds returning &null if the SHIFT key was pressed during the event reported by the most recent call of Event, otherwise fails.
`&x`	The x coordinate of the event reported by the most recent call of Event.
`&y`	The y coordinate of the event reported by the most recent call of Event.

17.9 Canvases and graphics contexts

There are two components of a window:

- the canvas, upon which the figures are drawn, and
- the graphics context, that specifies the font, line width, and other attributes that control the appearance of what is drawn in the window.

Some window attributes are canvas attributes, others are graphics context attributes.

The appearance of the window on the screen is controlled by canvas attributes. The attribute, `canvas`, controls window visibility. It's options are shown inTable 71.

Table 71 The canvas attribute.

`"canvas=normal"`	the window is visible and may be resized and moved about on the screen.
`"canvas=hidden"`	the window is invisible.
`"canvas=iconic"`	the window is represented by an icon.

Table 71 The canvas attribute.

`"canvas=maximal"`	the window occupies the entire screen.

If the window is `normal`, its position on the screen is determined by its `pos` (or `posx` and `posy`) attributes, its size by its `size` (or `height` and `width` or `lines` and `columns`) attributes. The window's label is the value of its `label` attribute. The window's initial contents can be specified on window creation by its `image` attribute. The window can be moved in front of overlapping windows by the function `Raise()`, or moved behind them with the function `Lower()`.

If the window is `iconic`, attribute `iconpos` specifies its position on the screen, `iconlabel` specifies the caption on its icon, and `iconimage` specifies the icon's graphics image.

There are a set of canvas attributes that identify the display screen the window appears on. Attribute `display` identifies the display device, `displayheight` gives its height in pixels, `displaywidth` gives its width, and `depth` gives the number of bits per pixel.

More than one window can share the same canvas. Function `Clone(W)` creates a new window with the same canvas as `W` but a different graphics context. Similarly, more than one window can share the same graphics attributes. Function `Couple(W1,W2)` creates a window combining W1's canvas and W2's graphics context.

Table 72. Canvas manipulation functions.

`Clone(` `  W,s1,...,sn)`	creates a new window with the same canvas as `W` but a different graphics context. The new graphics context is initialized from `W`'s and then is modified by the attribute assignments `s1`, ... ,`sn`.
`Couple(W1,W2)`	creates a window combining `W1`'s canvas and `W2`'s graphics context.
`Lower(W)`	moves the window, W, behind overlapping windows.
`Raise(W)`	moves the window, W, in front of overlapping windows.
`Uncouple(W)`	frees the window W. When the last binding to the same canvas is removed, the window is closed.
`WClose(W)`	closes window W: its canvas disappears from the screen. It still, however, exists and can be referenced via other bindings. Closing `&window` sets `&window` to `&null`.

17.10 Synchronizing window output

On some systems, the functions queue window output for display and return, rather than waiting for the output to be complete. This speeds up output, but can cause problems. The user's mouse commands may be responding to an earlier window contents. There are three functions to force the buffer to be written. They are shown in Table 73.

Table 73 Functions to flush the output buffer.

`WDelay(W,i)`	performs the rest of the output queued for window W, and then delays i milliseconds.
`WFlush(W)`	performs the rest of the window commands that have been queued for window W.
`WSync(W)`	waits until the rest of the window commands have been performed that have been queued for window W. WSync is aimed at client-server graphics.

17.11 Dialogs

There are several functions that perform standard dialogues with users. Four of them are shown in Table 74.

Table 74 Functions for standard dialogs.

`Dialog(`
`  W,L1,`
`  L2,L3,L4,`
`  L5,i)`
	displays a dialog box. List `L1` specifies the caption for the box, strings to be displayed one per line. List `L5` specifies one or more buttons: each string in `L5` specifies the caption on a button. Integer `i` is the index of the default button, or zero if there is no default.
	
	Lists `L2`, `L3`, and `L4` specify zero or more text entry fields: `L2[j]` gives the caption; `L3[j]`, the default values; and `L4[j]`, the maximum widths.
	
	Global variable `dialog_value` is assigned a list of the resulting text fields. The function returns the name of the button pressed to terminate the dialog.

`Notice(` `  W,s1,s2,...,sn)`	displays a dialog box with an "Okay" button. Each string `s`*i* is displayed on a different line. `Notice` returns the string "Okay" when the user presses the button.

Table 74 Functions for standard dialogs.

`OpenDialog(W,s1,s2)`	displays a dialog box containing a caption, and editable text string, and "Okay" and "Cancel" buttons. It is intended to be used for opening files. Parameter `s1` specifies the caption—`"Open:"` by default. Parameter `s2` specifies the initial value of the editable text string—the empty string by default. The edited text string value is placed in global variable `dialog_value`. `OpenDialog` returns the name of the button pressed.
`SaveDialog(W,s1,s2)`	displays a dialog box containing a caption, and editable text string, and "Yes", "No", and "Cancel" buttons. It is intended to be used for saving data in files. Parameter `s1` specifies the caption—`"Save:"` by default. Parameter `s2` specifies the initial value of the editable text string—the empty string by default. The edited text string value is placed in global variable `dialog_value`. `SaveDialog` returns the name of the button pressed.

Example. Here is an example using Notice.

```
link graphics
link dialog
procedure main()
   WOpen("size=500,500","canvas=normal",
     "pos=150,30") | stop("can't open window")
   Notice("Just give the word",
     "and I'll reformat",
     "your hard drive...")
   Notice("Just kidding")
end
```

17.12 Table of Attributes

`"ascent"`	the number of pixels the current font extends above the base line. Read only.

"bg=*color*"	selects the background color for the window. The default is "bg=white". Colors are presented in Section 17.6 on page 168.
"canvas=normal"	the window is visible and may be resized and moved about on the screen.
"canvas=hidden"	the window is invisible.
"canvas=iconic"	the window is represented by an icon.
"canvas=maximal"	the window occupies the entire screen.
"col=num"	column of the text cursor (in characters).
"cursor=*s*"	controls the visibility of the text cursor in the screen: "on" or "off", "off" by default.
"descent"	the number of pixels the current font extends below the base line. Read only.
"drawop=s"	string s specifies either "copy" or "reverse". With "copy" the new pixels replace the existing pixels. With "reverse", the new pixels will combine with the existing pixels in an exclusive-or fashion converting foreground-colored pixels into background-colored and vice versa.
"dx=*num*"	integer to be added to each x parameter passed to a graphics function to determine the actual pixel position in the window.
"dy=*num*"	integer to be added to each y parameter passed to a graphics function to determine the actual pixel position in the window.
"echo=*s*"	flag controlling whether characters typed to WRead and WReads are displayed on the screen: "on" or "off", "on" by default.
"fg=*color*"	selects the foreground color for the window. The default is "bg=black". Colors are presented in Section 17.6 on page 168.
"fheight"	height of the current text font, top to bottom. Read only.

`"fillstyle=s"`	string s specifies whether a pattern is to be used when drawing filled figures, and if so, how. The options are `"solid"`, `"textured"`, and `"masked"`. If `"solid"` is specified, figures are filled with the foreground color. If `"textured"` is specified, both the foreground and background colors of the pattern are used. If `"masked"`, only the foreground colors of the pattern are written; pixels corresponding to the background colors are left unchanged. The default is `"solid"`.
`"font=`*s*`"`	current text font.
`"fwidth"`	width of the current text font. Read only.
`"height=`*h*`"`	height of the window in pixels. Defaults to enough for 12 lines of text.
`"leading=`*num*`"`	the number of pixels between successive lines of text.
`"linestyle=`*s*`"`	string describing the style of the lines to be drawn. The options differ based on the underlying graphics system. For X-windows, the options are `solid`, `dashed`, and `striped`. The `dashed` lines have gaps. The `striped` lines are `dashed` with their gaps filled with the background color. In the beta version of Icon for Windows, the options are `dashed`, `dotted`, `solid`, `dashdotted`, and `dashdotdotted`. Style `striped` is treated as `dashed`. The default style is `solid`.
`"linewidth=`*num*`"`	number specifying the width of the lines drawn.
`"pattern=s"`	string s specifies the fill pattern to be used, either a pattern name (Figure 44) or a bi-level image (section 17.4.4).
`"pos=`*x*`,`*y*`"`	equivalent to: `"posx=`*x*`","posy=`*y*`"`. Defaults to `"pos=0,0"`, i.e. the window is drawn at the upper left corner of the screen.
`"posx=`*x*`"`	the horizontal position of the left edge of the window on the screen.

"posy=*y*"	the vertical position of the top edge of the window on the screen.
"reverse=s"	string s is either "on" or "off". When changed from "on" to "off" or "off" to "on", the values of bg and fg are swapped.
"row=*num*"	row of the text cursor (vertical position in characters).
"size=*w*,*h*"	equivalent to:"width=*w*","height=*h*"
"width=*w*"	width of the window in pixels. Defaults to enough for 80 columns of text.
"x=*num*"	horizontal position of the text cursor (in pixels).
"y=*num*"	vertical position of the text cursor (in pixels).

Chapter 18 Functions and keywords

Table 75 Summary of functions and keywords.

`abs(r)`	absolute value
`acos(r)`	arc cosine in radians, -1 ≤ r ≤ 1.
`Active()`	returns a window that has one or more events pending. It will wait until such a window is available. It will poll the open windows in a different order each time to avoid starvation. It fails if there are no open windows.
`addrat(r1,r2)`	Add rational numbers: r1+r2. `link rational`
`any(c)`	any(c,&subject,&pos,0)
`any(c,s)`	returns 2 if s[1] exists and is in character set c; otherwise it fails
`any(c,s,i)`	returns i+1 if s[i] exists and is in character set c; otherwise it fails
`any(c,s,i,j)`	returns i+1 if i<j and s[i] exists and s[i] is in character set c; otherwise it fails
`args(p)`	returns the number of parameters required by procedure p. If p is a user procedure with a variable number of parameters, `args(p)` returns the negative of the number of parameters p was declared with. If p is a built-in procedure with a variable number of parameters, `args(p)` returns -1.
`&ascii`	produces the character set containing all ASCII characters (128 characters).
`asin(r)`	arc sine in radians, -1 ≤ r ≤ 1.
`atan(r1,r2)`	arc tangent of r1/r2 in radians with the sign of r1.

Table 75 Summary of functions and keywords.

`atan(r)`	arc tangent of r in radians
`bal(c1,c2,c3)`	bal(c1,c2,c3,&subject,&pos,0)
`bal(c1,c2,c3,s)`	generates the positions k in s where 1≤k<*s+1 and s[k] (if it exists) is in cset c1, the number of characters in s[1:k] in cset c2 equals the number in c3, there is no position m, 1≤m≤k, where the number of characters in s[1:m] in cset c2 is less than the number in c3.
`bal(c1,c2,c3,s,i)`	generates the positions k in s where i≤k<*s+1 and s[k] (if it exists) is in cset c1, the number of characters in s[i:k] in cset c2 equals the number in c3, there is no position m, i≤m≤k, where the number of characters in s[i:m] in cset c2 is less than the number in c3.
`bal(c1,c2,c3,s,i,j)`	generates the positions k in s where i≤k<j and s[k] (if it exists) is in cset c1, the number of characters in s[i:k] in cset c2 equals the number in c3, there is no position m, i≤m≤k, where the number of characters in s[i:m] in cset c2 is less than the number in c3.
`basename(path, suffix)`	returns the base name of the file indicated by `path`. The `suffix` string is removed from the right. E.g. basename("D:\IPL\PROCS\BALQ.ICN", ".ICN") returns "BALQ". Works for UNIX, MSDOS, and MACs. `link basename`
`ceil(r)`	nearest integer to r away from 0. `link real2int`

Table 75 Summary of functions and keywords.

`center(s,i)`	produces a string of length i containing string s centered in it with blanks append to both sides to fill out the field. If *s>i, then it returns the middle i characters of s.
`center(s1,i,s2)`	produces a string of length i containing string s centered in it with copies of string s2 append to both sides to fill out the field. If *s>i, then it returns the middle i characters of s.
`char(i)`	produces a one character string where the single character has the internal representation given by integer i, 0(i(255.
`chdir(s)`	changes the current directory to that indicated by string s. Fails if it cannot change to that directory, perhaps because it does not exist.
`Clip(W,x,y,w,h)`	Set a clipping rectangle with its upper left corner at position (x,y), and width w and height h. The default w and h values are to the end of the window. Subsequent drawing outside the bounds will have no effect. When called without the (x,y) parameters, clipping is turned off and the entire canvas can be written. Returns the window, W.
`&clock`	returns a string giving the current time of day `"hh:mm:ss"`, in 24 hour form.
`Clone(` `  W,s1,...,sn)`	creates a new window with the same canvas as `W` but a different graphics context. The new graphics context is initialized from W's and then is modified by the attribute assignments s1, ... ,sn.
`close(f)`	closes the file bound to file object f.
`&col`	The text column of the event reported by the most recent call of Event.
`collect( )`	forces a garbage collection.
`collect(i)`	forces a garbage collection of region i, where i specifies the region 0—no specific region 1—static region 2—string region 3—block region

Table 75 Summary of functions and keywords.

`collect(i,j)`	forces a garbage collection of region i, where i specifies the region as shown for `collect(i)` above. Fails if there are not at least j bytes available in the region after collection.
`&collections`	generates four values: • the number of garbage collections • the number caused by attempts to allocate in the static region • the number caused by attempts to allocate in the string region • the number caused by attempts to allocate in the block region. The first value may be larger than the sum of the other three due to calls to `collect()`.
`Color(W,` `  n1,s1,n2,s2,...)`	sets each mutable color n_i to the corresponding color specified by s_i.

`components(s,sep)`
 	
`components(s)`
	returns a list of the components of the path `s`, where the components of the path are separated by the character `sep`. The separator defaults to "/" which is appropriate for UNIX. E.g.
	components("/a/b/c.d")
	returns
	["/","a","b","c.d"]
	`link filename`

`complex(r,i)`	create complex number with real part r and imaginary part i. `link complex`
`compress(s,c)`	Let x be a character in set c. A substring of s composed entirely of character x is replaced with a single character x. link strings
`&control`	succeeds returning &null if the CONTROL key was pressed during the event reported by the most recent call of Event, otherwise fails.

Table 75 Summary of functions and keywords.

`copy(x)`	`copy(x)` creates a new instance of any mutable object `x` (list, table, set, record) that has the same internal structure as `x`, but is not equal (`~===`) to `x`. Immutable objects like numbers, strings, csets are left as is.
`CopyArea(W1,W2, x1,y1,w,h,x2,y2)`	copy a rectangular area of width w and height h from position (x1,y1) in W1 to position (x2,y2) in window W2. The (x,y) values specify the upper-left corner, of course. If W2 is omitted, W1 is both the source and destination window. If W1 and W2 are both omitted, &window is the source and destination.
`cos(r)`	cosine of r (given in radians)
`Couple(W1,W2)`	creates a window combining W1's canvas and W2's graphics context.
`cpxadd(x1,x2)`	add complex numbers x1 and x2 `link complex`
`cpxdiv(x1,x2)`	divide complex number x1 by complex number x2 `link complex`
`cpxmul(x1,x2)`	multiply complex number x1 by complex number x2 `link complex`
`cpxsub(x1,x2)`	subtract complex number x2 from complex number x1 `link complex`
`cpxstr(x)`	convert complex number x to string representation `link complex`
`&cset`	produces the character set with all characters present (256 characters).
`&current`	the currently executing co-expression.
`&date`	returns the current date in the form `"yyyy/mm/dd"`.
`&dateline`	returns the current date and time of day as a string.
`decode(x)`	translates a string produced by encode back into a data structure isomorphic to the one encoded. `link codeobj`
`delete(S,x)`	deletes element x from set S. Returns set S.

Table 75 Summary of functions and keywords.

`delete(T,x)`	removes the key x and its value from table T.
`detab(s,i1,i2,...,in)`	copies string s replacing tab characters with blanks. The integer parameters give the tab stops. If more tab stops are needed, the last interval is repeated.
`Dialog(W, L1,L2,L3,L4,L5,i)`	displays a dialog box. List L1 specifies the caption for the box, strings to be displayed one per line. List L5 specifies one or more buttons: each string in L5 specifies the caption on a button. Integer i is the index of the default button, or zero if there is no default. Lists L2, L3, and L4 specify zero or more text entry fields: L2[j] gives the caption; L3[j], the default values; and L4[j], the maximum widths. Global variable dialog_value is assigned a list of the resulting text fields. The function returns the name of the button pressed to terminate the dialog.
`&digits`	'0123456789'
`display(i,f)`	writes to file f the names of the i most recently called active procedures, their local variables, and the global variables. Used for debugging.
`display(i)`	writes to `&errout` the names of the i most recently called active procedures, their local variables, and the global variables. Used for debugging.
`display()`	writes to `&errout` the names of all the active procedures, their local variables, and the global variables. Used for debugging.
`divrat(r1,r2)`	Divide rational numbers: r1 / r2. `link rational`
`dopen(s)`	opens the file named `s` using default options (i.e. `open(s,"rt")`). If the file is not found in the current directory, all the directories whose paths are listed in environment variable DPATH are tried, left to right until the file can be successfully opened. The paths in `DPATH` are separated from each other with blanks; the directories within the paths are separated by `"/"` characters. `link dopen`

Table 75 Summary of functions and keywords.

`dpath(s)`	returns the path for the file whose base name is s. If the file is not found in the current directory, all the directories whose paths are listed in environment variable DPATH are tried, left to right until the file can be successfully opened. The paths in `DPATH` are separated from each other with blanks; the directories within the paths are separated by "`/`" characters. (Icon on MS-DOS allows "`/`" rather than "`\`" in paths.) Procedure `dpath` returns • `s`, if the file is found in the current directory. • `path || "/" || s`, if the file is found at `path` within `DPATH`. `link dpath` or `link dopen` See also: `pathfind`.

`DrawArc(W,x,y,w,h)`
`DrawArc(W,`
`  x,y,w,h,theta,phi)`
	Draws an ellipse in the rectangle of width w and height h and a corner at position (x,y). If theta and phi are given, it draws an arc of the ellipse starting at angle theta and extending for angle phi.
	Returns the window, W.

`DrawCircle(W,x,y,r)`
`DrawCircle(W,`
`  x,y,r,theta,phi)`
	Draws a circle with center at position (x,y) and radius r. If theta and phi are given, it draws an arc of a circle starting at angle theta and extending for angle phi.
	Returns the window, W.

`DrawCurve(W,`
`  x1,y1,x2,y2,`
`  ...,xn,yn)`
	Draws a smoothed curve from (x1,y1) to (xn,yn) passing through the intermediate points in order. If xn=x1 and yn=y1, then the curve will be closed and smoothed through point (x1,y1) as well.
	Returns the window, W.

`DrawImage(W,x,y,s)`	will fill the rectangular area in window *`W`* with an upper left corner at position (*x*,*y*) with the image specified by string s. String s has one of three forms: *`"width,#hexdata"`* "*width*`,~hexdata`" *`"width,palette,chars"`* In all cases, the width *`width`* specifies the width of the rectangular area to fill. The height is determined by the number of characters or bits to write.

Table 75 Summary of functions and keywords.

`DrawImage(W,x,y, "width,#hexdata")`	the hexdata is a bi-level image.It is written as `textured`: the 1 bits are written as the foreground color and the 0 bits are written as the background color.
`DrawImage(W,x,y, "width,~hexdata")`	the hexdata is a bi-level image. It is written as `masked`: the 1's are written as foreground color but the 0's are not written, leaving those pixels unmodified.
`DrawImage(W,x,y, "width,palette,chars")`	the colors specified associated with the characters *chars* in palette *palette* are used to fill in the pixels from left to right, top to bottom. Character \377 and character ~ (if it is not contained in the palette) specify transparent pixels which will not overwrite the previous value. Commas and spaces, if the palette does not contain them, can be inserted to ease readability; they will not display.
`DrawLine(W, x1,y1,x2,y2, ...,xn,yn)`	Draws a connected sequence of straight lines. A straight line is drawn from point (x1,y1) to (x2,y2), then a line from (x2,y2) to (x3,y3), and so on to (xn,yn). Returns the window, W.
`DrawPoint(W, x1,y1,x2,y2,...,xn,yn)`	Draws a point at each position (x,y) given. Returns the window, W.
`DrawPolygon(W, x1,y1,x2,y2,...,xn,yn)`	Equivalent to DrawLine with a final line segment back to (x1,y1). Returns the window, W.
`DrawRectangle(W, x1,y1,x2,y2)`	Draws the rectangle with opposite corners (x,y) and (x+w,y+h). Parameters w and h default to the edge of the window. Returns the window, W.
`DrawSegment(W, x1,y1,x2,y2,...,xn,yn)`	Similar to DrawLine, except it draws disconnected line segments between pairs of points: (x1,y1) to (x2,y2), then (x3,y3) to (x4,y4), etc. Returns the window, W.
`DrawString(W,x,y,s)`	writes the string s starting at position (x,y) in window W without modifying the text cursor.
`dtor(r)`	degrees to radians

Table 75 Summary of functions and keywords.

`&e`	The base of the natural logarithms. Approximately 2.71828182845904
`encode(x)`	translates the data structure accessible from x into a string and returns that string. Any contained files, functions, procedures, co-expressions, and windows cannot be properly contained in a string, so don't try to include them. `link codeobj`
`Enqueue(W, e,x,y,cms,i)`	adds an event to the end of the list of pending events in window W. The event code is e. The position is (x,y). The string cms indicates the state of the control, meta, and shift keys: if the character "c" is in the string, the event reports the CONTROL key was pressed, and similarly "m" and "s" for META (ALT) and SHIFT. Parameter i gives the time in milliseconds for &interval. By default, e is &null, x is 0, y is 0, cms is "", and i is 0.
`entab(s,i1,i2,...,in)`	copies string s inserting tabs where possible. The integer parameters give the tab stops.
`EraseArea(W,x,y,w,h)`	Fills the rectangle with opposite corners (x,y) and (x+w,y+h) with the background color. Parameters w and h default to the edge of the window. Returns the window, W.
`&error`	controls whether errors cause the program to terminate. When zero, an error causes program termination with an error message. If nonzero, an error causes a failure and `&error` is decremented. The error message that would have been reported is instead assigned to keywords `&errornumber`, `&errortext`, and `&errorvalue`.
`errorclear()`	clears the indication that an error has occurred. References to the keywords `&errornumber`, `&errortext`, and `&errorvalue` fail until the next error has occurred.
`&errornumber`	the number of an error.
`&errortext`	the text explaining the error.
`&errorvalue`	the offending value (e.g., whose type didn't match). Access to `&errorvalue` will fail if there is no offending value associated with the error.

Table 75 Summary of functions and keywords.

`&errout`	the standard error output file. (It is not a variable; it cannot be reassigned.)
`Event(W)`	waits for an event in window W (if necessary), removes the first event from W's event queue, sets &control, &shift, &meta, &interval, &row, &col, &x, and &y, and returns the event code.
`exists(name)`	succeeds if file named `name` can be opened, otherwise fails. `link exists`
`exp(r)`	e^r, or in Icon, &e^(r)
`&features`	generates strings indicating the features of this version of Icon.
`&file`	the filename of the file this code was compiled from.
`FillArc(W,x,y,w,h)`	fills an ellipse within the rectangular area which has one corner at position (x,y) and width w and height h.
`FillArc(W, x,y,w,h,theta,phi)`	fills a wedge of an ellipse within the rectangular area which has one corner at position (x,y) and width w and height h. The wedge subtends the arc that would be drawn by DrawArc(`W,x,y,w,h,theta, phi`).
`FillCircle(W,x,y,r)`	fills a circle with center (x,y) and radius r.
`FillCircle(W, x,y,r,theta,phi)`	fills a wedge of a circle with center (x,y) and radius r. The wedge starts at angle theta and extends for angle phi.
`FillPolygon(W, x1,y1,x2,y2,...,xn,yn)`	fills the contained areas in the polygon that would be drawn by `DrawPolygon( W,x1,y1, x2,y2, ...,xn,yn)`
`FillRectangle(W, x,y,w,h)`	fills the rectangle with a corner at (x,y) and width w and height h.
`find(s1)`	find(s1,&subject,&pos,0)
`find(s1,s2)`	generates the positions k in s2 from 1 to *s2-*s1 which contain the beginning of the occurrences of s1, *i.e.*, where s2[k+:*s1]==s1. Fails if no occurrences of s1 are found.
`find(s1,s2,i)`	generates the positions k in s2 from i to *s2-*s1 which contain the beginning of the occurrences of s1, *i.e.*, where s2[k+:*s1]==s1. Fails if no occurrences of s1 are found.

Table 75 Summary of functions and keywords.

`find(s1,s2,i,j)`	generates the positions k in s2 from i to j-*s1 at which s1 occurs as a substring, *i.e.*, where s2[k+:*s1]==s1. Fails if no occurrences of s1 are found.
`findre(re,s,i,j)`	where re is a string containing the regular expression, and s, i, and j are as usual. To use it, see 6.13.1 on page 79. `link findre`.
`floor(r)`	nearest integer to r toward 0. `link real2int`
`flush(f)`	Output is typically buffered before being written. `flush(f)` flushes (actually writes out) the buffers for file `f`.
`Font(W,s)`	sets the font in window W to that specified by string s, or fails if it cannot be done.
`FreeColor(n1,...)`	frees the color map entries for all n_i. Not available in all implementations. Bad things may happen if any pixels are still assigned the color being freed.
`function()`	generates the names of Icon's built-in functions.
`gauss()`	returns a random number chosen from a gaussian distribution with a mean of zero `link gauss`
`gauss_random(x,f)`	returns a random number chosen from a gaussian distribution with a mean of x. Larger values of parameter f will flatten the distribution. `link gauss`
`gdl(dir)`	returns a list of all the file names in the directory indicated by the string `dir`. Fails if there are no files in the directory. Works with UNIX and MSDOS. Includes the directory in the file names. `link gdl2`
`gdlrec(dir)`	(recursive gdl) returns a list of all the file names in the directory indicated by the string `dir` and all its sub directories. Fails if there are no files in the directory. Works with UNIX and MSDOS. Includes the directory in the file names. `link gdl2`

Table 75 Summary of functions and keywords.

`get(L)`	removes and returns the first element of list L
`getch()`	reads a character from the keyboard, but does not echo it. Waits until a character is available.
`getche()`	reads a character from the keyboard and echoes it. Waits until a character is available.
`getenv(s)`	Systems typically provide environment variables: a table mapping string names into string values. `getenv(s)` returns the string associated with environment variable s, or fails if there is none such.
`getpaths(p1, p2,...,pn)`	generates `p1`, `p2`, ..., `pn` followed by all the paths in the `PATH` environment variable. This will work for both UNIX and MSDOS, choosing the correct PATH syntax for each.
`GotoRC(W,r,c)`	sets the text cursor in window W to row r and column c. GotoRC(W) sets the row and column to 1,1.
`GotoXY(W,x,y)`	sets the text cursor position in window W to position (x,y). GotoXY(W) sets the position to (0,0).
`&host`	the name of the computer system the program is running on.
`iand(i,j)`	bitwise **and**: a bit is set in the integer result only if it is set in both i and j.
`icom(i)`	bitwise **complement**: a bit is set in the integer result if and only if it is *not* set in i.
`image(s)`	produces a legible image of string s contained in double quotes. Characters \ and " are represented \\ and \". Special characters are represented in a form given in Table 11 on page 61, but if there is no *c* representation available, then the \x*hh* form is used.
`image(cs)`	produces a legible image of cset cs contained in single quotes. Characters \ and ’ are represented \\ and \’. Special characters are represented in a form given in Table 11 on page 61, but if there is no *c* representation available, then the \x*hh* form is used.
`image(n)`	produces the string representation of number n.

Table 75 Summary of functions and keywords.

`image(x)`	produces a legible image of object x. For the mutable objects, the general format is "*`type_num(size)`*", where *type* identifies the type of object, *num* identifies the particular instance of that type, and *size* gives the number of elements it contains.
`&input`	the standard input file. (It is not a variable; it cannot be reassigned.)
`insert(S,x)`	inserts element x into set S (if it is not already present). Returns S.
`insert(T,x,y)`	same as T[x]:=y, if T is a table.
`integer(x)`	converts x to an integer, if possible. Fails if not possible.
`&interval`	the interval, in milliseconds, between the event reported by the most recent call of Event and the event preceding it. This is not meaningful on all systems.
`ior(i,j)`	bitwise **or**: a bit is set in the integer result if it is set in either i or j.
`ishift(i,j)`	shift the bits in i by j positions to the left (if j>0) or |j| to the right (j<0), filling with zeros.
`ixor(i,j)`	bitwise **exclusive or**: a bit is set in the integer result only if it is set in one or the other but not both of i and j.
`kbhit()`	succeeds if a character has been typed at the keyboard that has not been read in yet. Use this to avoid waiting.
`key(T)`	generates all the keys in the table
`&lcase`	'abcdefghijklmnopqrstuvwxyz'
`&ldrag`	the integer event code returned when the mouse is moved while the left mouse button is pressed.
`left(s,i)`	produces a string of length i containing string s left justified with blanks append to the right to fill out the field. If *s>i, then it returns s[1:i+1]
`left(s1,i,s2)`	produces a string of length i containing string s left justified with copies of string s2 append to the right to fill out the field. If *s>i, then it returns s[1:i+1]
`&letters`	'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
`&level`	is the number of levels of active procedures calls.

Table 75 Summary of functions and keywords.

`&line`	is the number of the line this keyword occurs on.
`list()`	create an empty list
`list(n)`	create a list of n elements all initialized to `&null`
`list(n,val)`	create a list of `n` elements all initialized to the value of `val`
`log(r1,r2)`	logarithm of r1 to the base r2
`log(r)`	$\log_e$ r
`Lower(W)`	moves the window, W, behind overlapping windows.
`&lpress`	the integer event code returned when the left mouse button is pressed.
`&lrelease`	the integer event code returned when the left mouse button is released.
`&main`	the co-expression that initially began executing the program.
`many(c)`	many(c,&subject,&pos,0)
`many(c,s)`	returns the position in s following the longest initial substring in cset c. Returns *s+1 if all the characters are in c. Fails if the first character of s isn't in c. (This saves you from having to write something like: (upto(~c,s) | (any(c,s)&*s+1)) \1.)
`many(c,s,i)`	returns the position in s following the longest initial substring in cset c beginning at position i. Returns *s+1 if all the characters are in c. Fails if s[i] isn't in c.
`many(c,s,i,j)`	returns the position in s following the longest initial substring in cset c beginning at position i and not extending beyond position j. Returns j if all the characters are in c. Fails if s[i] is not in c or if s[i:j] would fail (*i.e.*, the range is not valid).
`map(s1,s2,s3)`	creates a new string which is a copy of s1 except for replacements made as follows: It replaces each character s1[i] that occurs in s2 at s2[j] with the character s3[j]. Strings s2 and s3 must be the same length. If the same character occurs more than once in s2, the rightmost occurrence determines the replacement character.

Table 75 Summary of functions and keywords.

`mapstrs(s,l1,l2)`	replaces substrings. Lists l1 and l2 contain strings. Each occurrence of a string of l1 in s is replaced. An occurrence of the ith string of l1 in s is replaced by the ith string in l2. If l2 is shorter than l1, the rightmost, unpaired strings in l1 are deleted. In cases of overlap, the leftmost match is preferred. If two strings match at the same location, the longer is preferred. link mapstrs
`match(s1)`	match(s1,&subject,&pos,0)
`match(s1,s2)`	returns *s1+1 if s2[1+:*s1] == s1; otherwise fails
`match(s1,s2,i)`	returns *s1+i if s2[i+:*s1] == s1; otherwise fails
`match(s1,s2,i,j)`	returns *s1+i if s2[i+:*s1] == s1; otherwise fails. Requires position j to be at least *s to the right of position i or it will fail.
`&mdrag`	the integer event code returned when the mouse is moved while the middle mouse button is pressed.
`member(S,x)`	succeeds if x is a member of set S, fails otherwise. Returns x if it succeeds.
`member(T,x)`	succeeds if key x is in table T. Returns x if it succeeds.
`&meta`	succeeds returning &null if the META (ALT) key was pressed during the event reported by the most recent call of Event, otherwise fails.
`move(i)`	moves `&pos` to position `&pos+i` in `&subject` and returns the substring between the original position of `&pos` and its new position. The new position can be zero or negative, but `&pos` is kept as a positive number. The assignment to `&pos` is reversible: when resumed during backtracking, `&pos` will be set back to its original position before the move.
`&mpress`	the integer event code returned when the middle mouse button is pressed.
`mpyrat(r1,r2)`	Multiply rational numbers: r1 * r2. `link rational`
`&mrelease`	the integer event code returned when the middle mouse button is released.

Table 75 Summary of functions and keywords.

`name(x)`	just as `image(x)` gives a legible indication of a value, `name(x)` gives a legible indication of a variable. If the variable, `x`, is a keyword or declared variable, name gives its name as a character string. If it is a component of a structure, `name(x)` gives the structure type (`list`, `table`,...) and the way the component is usually accessed, e.g., "`list[2]`", "`rec.f`".
`negrat(r)`	Negate a rational number: -r. `link rational`
`NewColor(W,s)`	allocates a changeable color map entry for a new mutable color and assigns it initially the color specified by s, and returns a small negative integer to represent the mutable color. Fails if no changeable color map entry is available.
`Notice(W,` `  s1,s2,...,sn)`	displays a dialog box with an "Okay" button. Each string `s`*`i`* is displayed on a different line. `Notice` returns the string "Okay" when the user presses the button.
`numeric(x)`	converts x to an integer or real value, if possible. Fails if not possible.
`open(s1,s2)`	opens the file named by string `s1` for access in the mode described by string `s2` and returns a file object that represented it, or fails if it cannot be opened. The modes are indicated by letters: "`a`"—open in append mode for writing "`b`"—open for both reading and writing "`c`"—create "`r`"—open for reading (default) "`w`"—open for writing "`t`"—translate line terminations into line feed characters (default)
`open(s1)`	is equivalent to `open(s1,"rt")`

Table 75 Summary of functions and keywords.

`OpenDialog(W, s1,s2)`	displays a dialog box containing a caption, and editable text string, and "Okay" and "Cancel" buttons. It is intended to be used for opening files. Parameter `s1` specifies the caption—`"Open:"` by default. Parameter `s2` specifies the initial value of the editable text string—the empty string by default. The edited text string value is placed in global variable `dialog_value`. OpenDialog returns the name of the button pressed.
`ord(s)`	converts the character in the one-character string s to its internal integer representation
`&output`	the standard output file. (It is not a variable; it cannot be reassigned.)
`PaletteChars(palette)`	returns as a character string the characters in the palette. For a grayscale palette, the characters will represent the intensities of gray from black to white in order. For uniform color palettes, cn, the first n^3 characters can be indexed by n intensity levels (0, 1, ..., n-1) for red, green, and blue, to select the character representing that combination of intensities, `P[r*n*n+g*n+b+1]`. The last n^2-2n+1 characters represent the additional gray levels in order from darkest to lightest.
`PaletteColor( palette,s)`	returns the color in palette represented by the single character s.
`PaletteGrays(palette)`	returns a string containing in order from black to white the characters from the palette representing levels of gray.
`PaletteKey(palette,s)`	returns a character from the palette which is close to the color specified by string s.

Table 75 Summary of functions and keywords.

`pathfind(s,p)`	returns the path for the file whose base name is `s`. If the file is not found in the current directory, all the directories whose paths are listed in string `p` are examined, left to right. If `p` is &null (*i.e.* not specified), the paths in environment variable DPATH are tried, left to right until the file can be successfully opened. The paths in `p` and `DPATH` are separated from each other with blanks; the directories within the paths are separated by "`/`" characters. (Icon on MSDOS allows "`/`" rather than "`\`" in paths.) Procedure `dpath` returns • `s`, if the file is found in the current directory. • `path || "/" || s` , if the file is found at `path` within `p` or `DPATH.` `link pathfind` See also: `dpath`.
`Pattern(W,s)`	establishes the pattern to be used for `Fill...` function calls. The pattern specification s may be one of the predefined names shown in Figure 44 on page 161, or it may be a bi-level image as described in Section 17.4.4 on page 161.
`pclose(file)`	closes the pipe bound to `file`, which was opened by `popen()`. `link popen`
`Pending(W)`	returns the list of events pending in window W. The list is empty is no events are pending.
`&phi`	phi, the "golden ratio," approx. 1.61803 = a/b where a/b=(a+b)/a
`&pi`	π, approximately 3.14159265358979
`Pixel(W,x,y,w,h)`	generates the colors of the pixels in the rectangle of width w and height h with its upper left corner in position (x,y) of window W. The pixels are generated by rows, left to right, top to bottom.
`pop(L)`	removes and returns the first element of list L

Table 75 Summary of functions and keywords.

`popen(s1,s2)`	equivalent to `open(s1,"p"||s2)` on systems with pipes. On systems without pipes, it will use the `system()` function and a temporary file to simulate a pipe. However, the command given in s1 will *not* run concurrently with the current process. (If you use `popen(s1, "w")`, you must use `pclose(file)` to actually have the command `s1` execute.) `link popen`
`proc(s)`	returns the procedure named s, where s is a string.
`proc(s,i)`	returns the procedure for the operator whose name is s which takes i parameters, e.g., proc("*",1)(x) *x proc("*",2)(x,y) x*y proc("[]",2)(x,y) x[y] proc("[:]",3)(x,y,z) x[y:z] proc("...",2)(x,y) x to y proc("...",3)(x,y,z) x to y by z
`prockind(x)`	fails if `x` is not a procedural value. Otherwise, it returns "`c`", if `x` is a record constructor, "`f`", if `x` is a built-in function, "`o`", if `x` is an operator, or "`p`", if `x` is a user-defined function. `link prockind`
`procname(x)`	returns the name of the procedure value `x` (which can also be a record constructor or operator), or fails if `x` is not a procedure. If `x` is an operator, its name has its number of parameters appended on the right, e.g. `procname(write)` yields "`write`" `procname(proc("...",3))` yields "`...3`" `link procname`
`&progname`	the file name of the executing program. It's a variable. You can assign another string to it if you wish.
`pull(L)`	removes and returns the last element of list L

Table 75 Summary of functions and keywords.

`push(L,x)`	inserts x as the new first element of list L, moving the other elements up one position, e.g., push([1,2,3],4) creates the same list as [4,1,2,3]
`push(L,x1,x2,...,xn)`	is equivalent to {`push(L,x1); push(L,x2); ... ; push(L,xn)`}. The end result is `xn` on top of the stack.
`put(L,x)`	inserts x as the new last element of list L, leaving the other elements in their previous positions, e.g., put([1,2,3],4) creates the same list as [1,2,3,4]
`put(L,x1,x2,...,xn)`	is equivalent to {`put(L,x1); put(L,x2); ... ; put(L,xn)`}.
`Raise(W)`	moves the window, W, in front of overlapping windows.
`&random`	The seed of the random sequence. You can assign a new value to it.
`randomize()`	a procedure to set the seed of the random number generator to a value determined in part from the date and time. You can use this to avoid always generating the same sequence of random numbers each time the program is run. `link randomiz`
`randreal(low,high)`	returns a random real number, r, in the range low ≤ r < high. `link randreal`
`ranseq(seed)`	*generates* the values of &random starting at seed. `link randseq`
`ranrange(min, max)`	returns a random integer in the range min to max, inclusive. `link ranrange`
`rat2str(r)`	Convert the rational number r to its string representation. `link rational`
`&rdrag`	the integer event code returned when the mouse is moved while the right mouse button is pressed.

Table 75 Summary of functions and keywords.

`read()`	reads and returns as a string the next line from the standard input file (`&input`), but fails on end of file. `read` strips off the terminating newline character from the line it returns.
`read(f)`	reads and returns as a string the next line from the file, `f`, but fails on end of file. `read` strips off the terminating newline character from the line it returns.

`ReadImage(W,s1,x,y)`
`ReadImage(W,`
`  s1,x,y,s2)`
	will read the image in the file named s1 into the window W with its upper-left corner at position (x,y). If s2 is specified, the colors in the image are converted into colors in palette s2.

`reads()`	reads and returns as a string the next character from the standard input file (`&input`), but fails on end of file.
`reads(f)`	reads and returns as a string the next character from the file, `f`, but fails on end of file.
`reads(f,i)`	reads and returns as a string the next i characters from the file, `f`. Fails on end of file. Returns fewer than i characters if only that many remain.
`real(x)`	converts x to a real number, if possible. Fails if not possible.
`reciprat(r)`	Get the reciprocal of rational number: 1/r. `link rational`
`ReFind(re,s,i1,i2)`	generates the positions in s of occurrences of regular expression re. The positions generated will be the leftmost positions of the matching strings. Regular expression re can be a string representation of a regular expression, or a list representation created by procedure RePat(s). See section 6.13.2 on page 80. link regexp
`&regions`	generates the current sizes of the three regions: static, string, and block. The size of the static region may not mean anything: Icon might allocate more space from the system when needed.

Table 75 Summary of functions and keywords.

`ReMatch(re,s,i1,i2)`	generates the positions in s following occurrences of regular expression re beginning at i1. Regular expression re can be a string representation of a regular expression, or a list representation created by procedure RePat(s). See section 6.13.2 on page 80. link regexp
`remove(s)`	removes the file named s from the disk directory, or fails if s cannot be removed.
`rename(s1,s2)`	renames the file whose name is s1 to have name s2. Fails if it cannot rename s1.
`RePat(s)`	translates a string representation of a regular expression into a list representation. If you are going to use the same expression repeatedly, it is best to translate it with RePat once rather than having ReMatch or ReFind translate the string representation repeatedly. See section 6.13.2 on page 80. link regexp
`repl(s,i)`	produces a string equal to i copies of s concatenated together
`replace(s1,s2,s3)`	replaces all occurrences of substring s2 in s1 by s3. link strings
`&resize`	the integer event code returned when the window is resized.
`reverse(s)`	produces the string s reversed
`right(s,i)`	produces a string of length i containing string s right justified with blanks append to the left to fill out the field. If *s>i, then it returns s[-i:0]
`right(s1,i,s2)`	produces a string of length i containing string s right justified with copies of string s2 append to the left to fill out the field. If *s>i, then it returns s[-i:0]
`round(r)`	nearest integer to r. `link real2int`
`&rpress`	the integer event code returned when the right mouse button is pressed.
`&rrelease`	the integer event code returned when the right mouse button is released.

Table 75 Summary of functions and keywords.

`rtod(r)`	convert radians to degrees
`runerror(i,x)`	cause the program to terminate with a standard run time error message for error number `i` and offending object `x`.
`save(s)`	saves the currently executing program as file s and returns the size of the file created. When executed, the program will resume executing by returning from the `save`. *Not available on all systems.*
`SaveDialog(W, s1,s2)`	displays a dialog box containing a caption, and editable text string, and "Yes", "No", and "Cancel" buttons. It is intended to be used for saving data in files. Parameter `s1` specifies the caption—`"Save:"` by default. Parameter `s2` specifies the initial value of the editable text string—the empty string by default. The edited text string value is placed in global variable `dialog_value`. SaveDialog returns the name of the button pressed.
`seek(f,i)`	seeks to position `i` in file `f` so that subsequent reads or writes will start at the i-th byte. Fails if the seek cannot be done. As in Icon strings, the first byte in the file is at position 1, and the last byte is indicated by position 0.
`segment(s,c)`	generates a sequence of strings which are the longest substrings of s from left to right composed solely of characters alternatively do or do not occur in c. link segment
`seq()`	generates the sequence 1,2,3,...
`seq(i)`	generates the sequence i,i+1,i+2,...
`seq(i,j)`	generates the sequence i,i+j,i+2j,...; j must not be 0.
`set()`	creates an empty set.
`set(L)`	creates a set whose initial contents are the elements of the list L.
`&shift`	succeeds returning &null if the SHIFT key was pressed during the event reported by the most recent call of Event, otherwise fails.
`sign(r)`	sign of r: -1 if r is negative, 0 if r is 0, 1 if r is positive. `link real2int`

Table 75 Summary of functions and keywords.

`sin(r)`	sine of r (given in radians)
`slashbal(` `  c1,c2,c3,s,i,j)`	like bal, but does not count a character from c2 or c3 that is preceded by a backslash character when determining balance. `link slashbal`
`slshupto(c,s,i,j)`	like upto, but treats backslash as an incorporation character in s, preventing the position of following character from being generated. Parameters s, i, and j default as in the built-in functions, but requires i≤j. (Warning: slshupto is reputed to have bugs.) `link slshupto`
`sort(L)`	creates a new list whose contents are the elements of list *L* in sorted order. Elements of the same type are grouped together. Lists, records, and other mutable objects are sorted in their group by their order of creation.
`sort(S)`	creates a list composed of the members of set S in sorted order. Elements of the same type are grouped together. Lists, records, and other mutable objects are sorted in their group by their order of creation.
`sort(T)`	(where T is a table) is the same as `sort(T,1)`
`sort(T,i)`	returns a list containing the keys and values from table T. If ki is the ith key and vi is its corresponding value, the resulting list is: [[k1,v1],[k2,v2],...[kn,vn]] sorted by keys if i=1 [[k1,v1],[k2,v2],...[kn,vn]] sorted by values if i=2 [k1,v1,k2,v2,...kn,vn] sorted by keys if i=3 [k1,v1,k2,v2,...kn,vn] sorted by values if i=4
`sortf(L,i)`	creates a new list whose contents are the elements of list *L* in sorted order. Records and lists contained in *L* with a size of at least *i* are sorted by their *i*th field.
`sortf(S,i)`	creates a new list whose contents are the elements of set *S* in sorted order. Records and lists contained in *S* with a size of at least *i* are sorted by their *i*th field.
`&source`	the co-expression that activated the current co-expression.

Table 75 Summary of functions and keywords.

`sqrt(r)`	square root of real r ≥ 0.
`stop(x1,x2,...)`	writes out the values `x1,x2,...` left-to-right to the error output, `&errout`, and exits with an error status. If any `xi` is a file, subsequent output is to that file.
`&storage`	generates the amount of space currently in use in the three regions: static, string, and block. Again, the space occupied in the static region may not mean anything.
`str2rat(s)`	Convert the string representation of a rational number (such as "3/2") to a rational number. `link rational`
`strcpx(s)`	convert string representation s of a complex number to it's internal representation `link complex`
`string(x)`	converts a number or a cset to a string.
`subrat(r1,r2)`	Subtract rational numbers: r1 - r2. `link rational`
`suffix(s,sep)` `suffix(s)`	returns the list `[pre,post]` where `pre` is the substring of `s` up to the last occurrence of `sep` and `post` is the substring of `s` to the right of the `sep`. The separator defaults to `"."`, appropriate for both UNIX and MSDOS. If the separator `sep` does not occur, `suffix` returns `[s,&null]`. `link filename`
`system(s)`	executes the string `s` as a system (shell) command and returns the exit status (an integer) by calling the C function `system`. It is not available on all systems, but it is available on UNIX©. The command should be able to direct its output to a file that the program can then open and read.
`tab(i)`	moves `&pos` to position `i` in `&subject` and returns the substring between the original position of `&pos` and its new position. Position `i` can be zero or negative, but `&pos` is kept as a positive number. The assignment to `&pos` is reversible: when resumed during backtracking, `&pos` will be set back to its original position before the `tab`.

Table 75 Summary of functions and keywords.

`table( )`	returns a new table. Attempting to look up a key not in the table returns `&null`. For example, `t[[]]` will yield `&null` because the new list `[]` can't be in the table.
`table(x)`	returns a new table. Attempting to look up a key not in the table returns the value of `x`. For example, `t[[]]` will yield the value of `x` because the new list `[]` cannot be in the table. The expression `x` is evaluated when the table is created, so if you execute `t:=table([])`, all new keys you look up will point to the *same* list.

`tail(s,sep)`
 	
`tail(s)`
	returns the list `[pre,post]` where `pre` is the substring of `s` up to the last occurrence of `sep` and `post` is the substring of `s` to the right of the separator `sep`. The separator defaults to "`/`" which is appropriate for UNIX paths. Since Icon allows MSDOS paths to be specified with "/" rather than "\", it can be used for DOS if you translate the paths. There are a number of special cases, `tail` returns
	• `["",s]` if `sep` does not occur in `s`.
	• `[sep,s[2:0]]` if `sep==s[1]`.
	• `[s[1:j],s[j+1:0]]` if `sep` occurs at position `j`, `1<j<*s-1`.
	• `[s[1:-1],&null]` if `sep` occurs as the last character in `s`.
	
	`link filename`

`tan(r)`	tangent of r (given in radians)

Table 75 Summary of functions and keywords.

`tempname()`	generates names for a temporary file, *i.e.* a file that does not appear to already exist. Under UNIX, the file name has the form
	`/tmp/icontmp.`*ddd*
	where *ddd* is a string of exactly three digits. Under MS-DOS, the filename is either of the forms:
	temp`\icon0`*ddd*`.tmp`
	or
	`icon0`*ddd*`.tmp`
	The first form uses the directory bound to the environment variable TEMP. If TEMP is not defined, then the second form is used, placing the file in the current directory.
	
	Because Icon cannot directly test whether a file exists, `tempname` returns the names of files it could not open for reading, which might mean the file exists but is locked. In that case, you will not be able to open it for writing either. Therefore `tempname` is a generator so that if you can not open the first file generated, you should be able to open a subsequent one.
	
	`link tempname`

`TextWidth(W,s)`	returns the number of pixels of width that string s would require if written in window W.
`&time`	returns the number of milliseconds since the program started executing.
`&trace`	when not equal to zero, every procedure call, return, suspension, or resumption writes a message to `&errout` and decrements `&trace`.
`trim(s)`	produces a copy of string s with trailing blanks removed
`trim(s,cs)`	produces a copy of string s with all the rightmost characters that are contained in cset cs removed
`trunc(r)`	nearest integer less than r. `link real2int`

`type(x)`	produces a string naming the type of object s, one of:
	
	`"integer" "real" "string"`
	`"cset" "list"`
	`"table" "set" "procedure"`
	`"co-expression" "window"`
	or the name of a record type.

Table 75 Summary of functions and keywords.

`&ucase`	'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
`Uncouple(W)`	frees the window W. When the last binding to the same canvas is removed, the window is closed.
`upto(c)`	upto(c,&subject,&pos,0)
`upto(c,s)`	generates the positions in s from 1 to *s which contain characters in set c. Fails if no such character is found.
`upto(c,s,i)`	generates the positions in s from position i to *s which contain characters in set c. Fails if no such character is found.
`upto(c,s,i,j)`	generates the positions in s from position i to position j which contain characters in set c. Fails if no such character is found.
`variable(s)`	returns the variable or variable keyword whose name is contained in string s. It will only return a variable known at the place of call—you can only access a local variable within a procedure.
`&version`	is a string representation of the version of Icon that is executing.
`WAttrib(W, s1,s2,...)`	sets and queries the attributes of a window. Each string is either "*name*" or "*name=value*" where name is the name of an attribute and value is a string representation of a value for that attribute. First WAttrib will perform assignments for all the "*name=value*" parameters, then it will generate the values for all the attributes named, left to right. Generates the values of the attributes. Fails on an attempt to set an invalid `bg`, `fg`, `font`, or `pattern`. Gives a run-time error on any other invalid value or name.
`WClose(W)`	closes the window W. The window is removed from the screen. It still, however, exists and can be referenced via other bindings. Closing `&window` sets `&window` to `&null`.
`WDelay(W,i)`	performs the rest of the output queued for window W, and then delays i milliseconds.
`WDone(W)`	waits until a `Q` (or `q`) is typed in the window, then closes it.

Table 75 Summary of functions and keywords.

`WFlush(W)`	performs the rest of the window commands that have been queued for window W.
`where(f)`	returns the current file position, most likely for use with `seek` later.
`WOpen( s1,s2,...,sn)`	opens a new window with the values of its attributes given by the strings. Each string is of the same form as a value assignment in `WAttrib`: "*name=value*" where name is the name of an attribute and value is a string representation of a value for that attribute. Returns the window. Assigns the window to `&window` if `&window` is previously `&null`.
`WRead(W)`	reads a line typed into window W in the manner of `read`. Displays the text cursor and echoes the characters typed if window attributes `cursor` and `echo` allow it.
`WReads(W,i)`	reads i characters typed into window W in the manner of `reads`. Displays the text cursor and echoes the characters typed if window attributes `cursor` and `echo` allow it.
`write(x1,x2,...)`	writes out the values `x1,x2,...` left-to-right to the standard output, and follows them with a line termination. If any `xi` is a file, the following values are written to that file until the file is changed again or the end of the write procedure. If any xi is neither a file nor a string and cannot be converted to a string, `write` terminates program execution with an error.
`WriteImage(W, s,x,y,w,h)`	writes the rectangle of pixels of width w and height h with its upper left corner in position (x,y) of window W into the file named s. Normally Icon writes the file in GIF format, but it may allow other extensions on the file name to choose other formats.
`writes(s1,s2,...)`	writes out the values `x1,x2,...` left-to-right to the standard output. It does *not* follow them with a line termination. If any `xi` is a file, the following values are written to that file until the file is changed again or the end of the write procedure. If any xi is neither a file nor a string and cannot be converted to a string, `writes` terminates program execution with an error.
`WSync(W)`	waits until the rest of the window commands have been performed that have been queued for window W. WSync is aimed at client-server graphics.

Table 75 Summary of functions and keywords.

`WWrite(W, s1,s2,...,sn)`	writes the strings s1, s2, .., sn in window W in the manner of `write`—followed by moving the cursor to the beginning of the next line, scrolling if required.
`WWrites(W, s1,s2,...,sn)`	writes the strings s1, s2, .., sn in window W in the manner of `writes`—scrolling if required by any \n characters written.
`&x`	The x coordinate of the event reported by the most recent call of Event.
`xdecode(f)` `xdecode(f,p)`	reads, reconstructs, and returns the Icon data structure from file `f` that was previously saved there by xencode. Files, co-expressions, and windows are decoded as empty lists (except for files `&input`, `&output`, and `&errout`). Fails if the file is not in xcode format or if it contains an undeclared record. If p is provided, `xdecode` reads the lines calling `p(f)` rather than `read(f)`. See `xencode` for an idea of what to use this for. `link xcode`
`xdecoden(x,fn)`	like `xdecode`, except that `fn` is the name of a file to be opened for input (with `open(fn)`). `link xcode`

`xencode(x,f)`
`xencode(x,f,p)`
	encodes and writes the data structure `x` into file `f`. The data structure can be read back in by `xdecode`. If parameter p is provided, it is called in place of `write`, *i.e.* `p(f,...)` instead of `write(f,...)`, in which case `f` need not be a file, *e.g.*
	`xencode(x,L:=[],put)`
	will encode the data structure into a list, `L`.
	
	`link xcode`

`xencoden(x,fn,opt)`	like `xencode`, except that `fn` is the name of a file to be opened for output (with `open(fn, opt)`). The options, `opt`, default to `"w"`. `link xcode`
`&y`	The y coordinate of the event reported by the most recent call of Event.

Chapter 19 Syntax

19.1 Grammar for Icon

In the grammar, all literal characters are quoted. The equal sign defines the name on its left hand side to match the pattern on its right. The vertical bar separates alternatives. Parentheses' group alternatives. Brackets enclose things that may or may not be present. Braces enclose things that may be present any number of times or may be absent entirely.

```
start : program.
program = declaration
     | declaration program
     .
endOfExpr = ";" | EOL .
declaration = link_declaration
     | global_declaration
     | record_declaration
     | procedure_declaration
     .
link_declaration = "link" link_list
     .
link_list = file_name
     | file_name "," link_list
     .
file_name = identifier
     | string_literal
     .
global_declaration = "global" identifier_list
     .
identifier_list = identifier
     | identifier_list "," identifier
     .
record_declaration =
       "record" identifier "("
          field_list_opt
       ")"
     .
field_list_opt = field_list
     |
     .
```



```
field_list = field_name
     | field_list "," field_name
     .
field_name = identifier
     .
procedure_declaration =
       proc_header
       locals_opt
       initial_opt
       expression_sequence
       "end"
     .
proc_header =
       "procedure" identifier
          "(" parameter_list_opt ")" endOfExpr
     .
parameter_list_opt = parameter_list
     |
     .
parameter_list = identifier
     | identifier "[" "]"
     | identifier "," parameter_list
     .
locals_opt = locals
     |
     .
locals = local_specification identifier_list
     | local_specification
          identifier_list endOfExpr locals
     .
local_specification = "local"
     | "static"
     .
initial_opt = "initial" expression endOfExpr
     |
     .
expression_sequence = expression_opt
     | expression_sequence endOfExpr expression_opt
     .
expression_opt = expression
     |
     .
expression =
       "break" expression_opt
     | "create" expression
     | "return" expression_opt
     | "suspend" expression_opt
          suspend_do_clause_opt
     | "fail"
```



```
     | "next"
     | "case" expression "of" "{"
           case_list
       "}"
     | "if" expression "then" expression
           else_clause_opt
     | "repeat" expression
     | "while" expression while_do_clause_opt
     | "until" expression until_do_clause_opt
     | "every" expression every_do_clause_opt
     | expr1
     .
suspend_do_clause_opt = "do" expression | .
while_do_clause_opt = "do" expression | .
until_do_clause_opt = "do" expression | .
every_do_clause_opt = "do" expression | .
else_clause_opt = "else" expression | .
case_list = case_clause
     | case_list endOfExpr case_clause
     .
case_clause = expression ":" expression
     | "default" ":" expression
     .
expr1= expr1 "&" expr2
     | expr2
     .
expr2= expr2 "?" expr3
     | expr3
     .
expr3=   expr4 ":=" expr3
     | expr4 ":=:" expr3
     | expr4 "<-" expr3
     | expr4 "<->" expr3
     | expr4 op_asgn expr3
     | expr4
     .
expr4= expr4 "to" expr5
     | expr4 "to" expr5 "by" expr5
     | expr5
     .
expr5= expr5 "|" expr6
     | expr6
     .
expr6= expr6 "<" expr7
     | expr6 "<=" expr7
     | expr6 "=" expr7
     | expr6 ">=" expr7
     | expr6 ">" expr7
     | expr6 "~=" expr7
```



```
        | expr6 "<<" expr7
        | expr6 "<<=" expr7
        | expr6 "==" expr7
        | expr6 ">>=" expr7
        | expr6 ">>" expr7
        | expr6 "~==" expr7
        | expr6 "===" expr7
        | expr6 "~===" expr7
        | expr7
        .
expr7=  expr7 "||" expr8
        | expr7 "|||" expr8
        | expr8
        .
expr8=  expr8 "+" expr9
        | expr8 "-" expr9
        | expr8 "++" expr9
        | expr8 "--" expr9
        | expr9
        .
expr9=  expr9 "*" expr10
        | expr9 "/" expr10
        | expr9 "%" expr10
        | expr9 "**" expr10
        | expr10
        .
expr10= expr11 "^" expr10
        | expr11
        .
expr11= expr11 "\" expr12
        | expr11 "@" expr12
        | expr11 "!" expr12
        | expr12
        .
expr12= "not" expr12
        | "|" expr12
        | "!" expr12
        | "*" expr12
        | "+" expr12
        | "-" expr12
        | "." expr12
        | "/" expr12
        | "\" expr12
        | "=" expr12
        | "?" expr12
        | "~" expr12
        | "@" expr12
        | "^" expr12
        | expr13
```



```
        .
expr13= "(" expression_list ")"
      | "{" expression_sequence "}"
      | "[" expression_list "]"
      | expr13 "." field_name
      | expr13 "(" expression_list ")"
      | expr13 "{" expression_list "}"
      | expr13 "[" subscript_list "]"
      | identifier
      | keyword
      | literal
        .
expression_list = expression_opt
      | expression_list "," expression_opt
        .
subscript_list = subscript
      | subscript_list "," subscript
        .
subscript = expression
      | expression ":" expression
      | expression "+:" expression
      | expression "-:" expression
        .
keyword = "&" identifier
        .
literal = string_literal
      | integer_literal
      | real_literal
      | cset_literal
        .
```

19.2 Table of operators

Icon has many operators and many precedence levels. We include showing the operators from highest precedence to lowest for you to refer back to when you need it.. However, it is much safer to use parentheses liberally than to try to remember the precedence levels.

precedence	associativity	numeric	string	cset	list	set	co-expression	other
12	unary	+ - ?	* ? ! =	~ *	* ? !		@ ^ *	| not . / \
11	binary left						@	! \
10	binary right	^						
9	binary left	* / %		**		**		

8	binary left	`+ -`		`++ - -`		`++ --`		
7	binary left		`||`		`|||`			
6	binary left	`< <= > >= = ~=`	`<< <<= >> >>= == ~==`					`=== ~===`
5	binary left							`|`
4	left, binary or ternary							`e1 to e2` `e1 to e2 by e3`
3	binary right							`:= :=: <- <-> op:=`
2	binary left		`?`					
1	binary left							`&`

Chapter 20 Bibliography

Foley, James D., Andries van Dam, Steven K. Feiner, John F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company, 1990.

Griswold, Ralph E. and Madge T. Griswold, *The Icon Programming Language*, Third Edition, Peer-to-Peer Communications, 1996.

Griswold, Ralph E. and Madge T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.

Jeffery, Clinton L., Gregg M. Townsend, and Ralph E. Griswold, *Graphics Facilities for the Icon Programming Language: Version 9.0*, IPD255, The Icon Project, University of Arizona, July 19, 1994.

Griswold, Ralph E., *Version 9.0 of the Icon Compiler*, IPD237, Icon Project, Department of Computer Science, University of Arizona, May 1994.

