User's Manual for the Icon Programming Language*

Ralph E. Griswold

TR 78-14

Department

of

Computer Science

The University of Arizona

User's Manual for the Icon Programming Language*

Ralph E. Griswold

TR 78-14

October 6, 1978

The Department of Computer Science
The University of Arizona

*This work was supported in part by the National Science Foundation under Grant MSC75-01307.

CONTENTS

Chapter 4 -- String Processing

Chapter 5 -- Structures

CHAPTER 1

Introduction

This manual describes the Icon programming language. It is neither a tutorial nor a detailed reference manual. It attempts to give a comprehensive coverage of the language in a complete but informal way. The reader is assumed to have experience with other programming languages. A familiarity with SNOBOL4 [1] will be helpful in placing the concepts in perspective.

The first part of this manual gives an overview of Icon and presents the techniques that are used for describing language features. Subsequent chapters describe the language in detail. There are a number of appendices at the end of this manual that provide quick reference to frequently needed information.

1.1 An Overview of Icon

Icon is a general-purpose programming language with an emphasis on string processing. Icon is a descendant of SNOBOL4 and SL5 [2] and shares much of the philosophical bases of these languages.

Icon differs from SNOBOL4 in that it provides string processing that is integrated into the language rather than as a separate pattern-matching facility. Icon lacks some of the exotic features of both SNOBOL4 and SL5; in order to provide greater efficiency in the most frequently used operations, Icon restricts run-time flexibility. In this sense, Icon follows the more traditional method of binding many language operations at compile time.

One of the unusual characteristics of Icon is goal-directed expression evaluation, which provides automatic searching for alternatives and a controlled form of backtracking. This method of evaluation allows concise, natural formulation of many algorithms while avoiding the inefficiency of uncontrolled backtracking.

Syntactically, Icon is a language in the style of Algol 60. It has an expression-based structure and uses reserved words for many constructs.

In addition to conventional control structures, Icon has a number of unusual control structures related to alternatives and goal-directed evaluation. The result of expression evaluation is both a value and a signal. The signal indicates the success or failure of the operation (as in SNOBOL4 and SL5) and is used to drive control structures.

To balance efficiency and ease of use, Icon provides optional compile-time specifications and defaults. Variables may be typed or nontyped. Nontyped variables are treated as in SNOBOL4 and SL5, with automatic type checking and coercion. On the other hand, variables may be typed to provide error checking and greater efficiency. Similarly, there are default values for the arguments of many functions, allowing conciseness and suppression of notational detail.

1.2 Syntax Notation

The syntax of Icon is described in a semi-formal manner with emphasis on clarity rather than rigor. For simple cases, English prose is generally used. Where the syntax is more complicated, a formal metalanguage is used.

In this metalanguage, syntactic classes are denoted by italics. For example, *expr* denotes the class of expressions. The names of the syntactic classes are chosen to be mnemonic, but have no formal significance. Program text is given in the regular type face, except for reserved words, which are given in boldface for emphasis. There is, of course, no distinction between reserved words and other program text in actual programs, except for the significance of the reserved word names.

Alternatives are separated by bars (|) and by the vertical stacking of items. Braces ({}) enclose mandatory items, while brackets ([]) enclose optional items. Ellipses (...) indicate indefinite repetition of items. The metalinguistic and literal uses of bars, brackets, braces and periods are not mixed in any one usage, and the meaning should be clear in context. In the summary of the syntax given in Appendix A, ambiguity is resolved by using primitive syntactic classes. For example, *bar* denotes the symbol | and the symbol [is denoted by *left-bracket*.

1.3 Program Text and Character Sets

The natural character set for Icon is ASCII [3]. To allow for compatibility with computers and equipment that do not support the full ASCII character set, the following characters are equivalent syntactically:

```
lower-case letters and upper-case letters
blank and tab
^ and ~
` and @
[ and {
] and }
| and \ and !
```

In the program examples given in this manual, lower-case letters are used exclusively. However, these letters can be entered in upper case in a program without changing the operation of the program. Similarly, braces and brackets are used differently in

the manual, although they can be used interchangeably in a program. Bars and backslashes are treated in the same fashion. The important point is that the equivalences above apply uniformly to program constructs (although characters in literals are taken exactly as they are entered).

1.4 Organization of the Manual

This manual is organized around chapters describing the major features of the language. For example, all the string-processing operations are described in one chapter. Each operation and function is described separately or is grouped with others of a similar nature. Following the description, examples of usage are given.

The examples are not intended to motivate language features, but rather to provide concrete instances, to show special cases that may not be clear otherwise, and to illustrate possibilities that may not be obvious. For these reasons, many of the examples are contrived and are not typical of ordinary usage.

Where appropriate, there are remarks that are subsidiary to the main description. These remarks are divided into notes, warnings, defaults, failure conditions, and error conditions. The notes describe special cases, details, and such. The warnings are designed to alert the programmer to programming pitfalls and hazards that might otherwise be overlooked. The defaults describe interpretations that are made in the absence of optional parts of expressions. The failure conditions specify situations in which an operation may signal failure. The error conditions specify situations that are erroneous and cause program termination. The defaults and error conditions are summarized in Appendices C and E.

It is not always possible to describe language features in a linear fashion; some circularity is unavoidable. This manual contains numerous cross references between sections. In the case of forward references, an attempt has been made to make the referenced items clear in context even if they cannot be completely described there. For a full set of references, see the index.

CHAPTER 2

Basic Concepts

2.1 Values and Types

Computation involves the specification, creation, and comparison of data. The concept of value is a fundamental one. The nature of values varies from one kind of data to another and much of this manual is concerned with various kinds of values.

Icon supports several kinds of data, called types:

integer
real
string
cset
file
procedure
array
table
stack
null

Integers and reals (floating-point numbers) serve their conventional purposes. Strings are sequences of characters as in SNOBOL4. Csets are sets of characters in which membership is significant, but order is not. Files identify external data storage. Procedures serve their conventional purpose, but it is notable that they are data objects. Arrays, tables, and stacks are data structures with various organizations and access methods. The null type serves a special purpose as an identity object and is convertible to other types. For example, the integer equivalent of &null is 0, while the string equivalent of &null is the string containing no characters. In addition to the types listed above, there is a facility for defining record types. The type names are reserved words that have different roles, depending on context.

Types are indicated in examples by letters related to conventional usage or the type name. In particular, i, j, and k are used to indicate integers, while s1, s2, and s3 are used to indicate strings and x indicates an object of undetermined type.

The values of certain types can be specified literally in the program text. These are **integer**, **real**, and **string**. Integers and reals are represented as constants in the conventional manner. For example, 300 is an integer, while 1.0 is a real. Strings are enclosed in quotation marks, as in "summary". See Sections 3.1.1, 3.2.1, and 4.2.1 for further descriptions of the methods available for representing literals. Values of types other than

these can be constructed and computed in a variety of ways, but they do not have literal representations.

2.2 Variables

A variable is an entity that can have a value. Variables provide a way of storing and referencing values that are computed during program execution.

The simplest kind of variable is the identifier. Syntactically, an identifier must begin with a letter, which may be followed by any number of other letters and digits. Underscores may occur within an identifier but not at the beginning or end. Reserved words may not be used as identifiers.

examples:

syntactically correct identifiers

```
x
X
k00001
summary
report1
node_link
```

syntactically erroneous identifiers

```
23K
report$
x0"r
string
node_
_link
```

There are various forms of variables other than identifiers. Some variables, such as the elements of an array, are computed during program execution and have various syntactic representations. See Sections 4.5.4, 5.1.2, 5.2.2, 5.3.2, 5.4.3, and 8.2.2.

2.3 Assignment

One of the most fundamental operations is the assignment of a value to a variable. This operation is performed by the := infix operator. For example,

```
x := 3
```

assigns the integer value 3 to the identifier x.

Note: The assignment operator associates to the right and returns the value of its right operand. Thus multiple assignments can be made as in the following example

```
x := y := 3
```

which assigns 3 to both x and y.

Any variable may appear on the left side of an assignment operation and any expression may appear on the right. For example,

```
x := z
```

assigns the value of the identifier z to the identifier x.

Error Condition: If the expression on the left side of the assignment operation is not a variable, Error 121 occurs.

The infix operator :=: exchanges the values of its operands. For example,

```
x :=: y
```

exchanges the values of x and y.

Note: The exchange operator associates to the right and returns the value of its right operand.

Error Condition: If the expression on either side of the exchange operation is not a variable, Error 121 occurs.

2.4 Keywords

Keywords provide an interface between the executing program and the environment in which it operates. Keywords have a number of uses. Some have important constants as values, others change the status of global conditions, while others provide the values of environmental variables.

Keywords are distinguished by an ampersand followed by a word that has a special meaning. Examples are &date, whose value is the current date, and &null, whose value is the object of type **null**.

Some keywords are variables, and values can be assigned to them to set the status of conditions. An example is &trace, which controls the tracing of procedure calls (see Section 8.2.4). If &trace is assigned a nonzero value, tracing is enabled, while a zero value disables tracing.

Some keywords are not variables and cannot be assigned values. An example is &date.

Keywords are described throughout this manual in the sections that relate to their use.

2.5 Built-in Functions

Built-in functions provide much of the computational repertoire of Icon. Function calls have a conventional syntax in which the function name is followed by arguments in an expression list enclosed in parentheses:

name (expr [, expr] ...)

For example, size(x) produces the size of object x, map(s1,s2,s3) produces a character mapping on s1, and write(s) writes the value of s.

As indicated, an argument may be any expression of arbitrary complexity.

Different functions expect arguments of different types, as indicated above. Automatic conversion (coercion) is performed to convert arguments of other types to the expected types.

Error Condition: If an argument cannot be converted to a required type, an error with a number of the form 10n occurs, where n is a digit that identifies the expected type. See Appendix E.

Default: Omitted arguments default to &null and are converted to the required type unless otherwise noted. In some cases, omitted arguments have special defaults. These cases are noted throughout the manual and are summarized in Appendix C. If trailing arguments are omitted, the trailing commas may be omitted also.

Failure Condition: As indicated in Section 1.1, some functions fail under certain conditions. See also Section 2.7. If the evaluation of an argument fails, the function is not called, and the calling expression fails. If more arguments are provided than are required by the function, the extra arguments are evaluated, but their values are ignored. If an extra argument fails, however, the function is not called and the calling expression fails.

2.6 Operators

Operators provide a convenient abbreviated notation for functions. There are three kinds of operators: prefix, infix, and suffix.

Prefix operators have one operand (argument). An example is -i, which produces the negative of i. Suffix operators also have one operand, but follow rather than precede the operand. An example is i-, which decrements the value of i by one and produces the new value.

Note: The & in keywords is part of the keyword and is not a prefix operator.

Infix operators have two operands and stand between them. Examples are i + j and i * j, which produce the sum and product of i and j, respectively.

Failure Condition: If evaluation of an operand of an operation fails, the operation is not performed and the expression fails.

While all prefix and suffix operators are single symbols, some infix operators are composed of more than one symbol. Examples are i ** j, which produces i raised to the power j, s1 || s2, which produces the concatenation of the strings s1 and s2, and s1 == s2, which compares strings s1 and s2 for equality.

Various operators used in conjunction produce potentially ambiguous expressions. For example, i--j might be interpreted in several ways. Blanks may be used to differentiate otherwise ambiguous expressions. For example, i- - j and i - -j are clearly different. Parentheses may also be used for grouping. The expressions (i-)-j and i-(-j) are alternate forms of those given above.

In the absence of blanks or parentheses, rules are used to interpret potentially ambiguous expressions. In addition, rules of precedence and associativity are used to determine which operands are associated with which operators in complex expressions.

As a class, prefix operators have the highest precedence (bind most tightly to their operands). Suffix operators have the next highest precedence, and infix operators have the lowest precedence. For example, -i*j is equivalent to (-i)*j, while i*j- is equivalent to i*(j-). Different infix operators have different precedences. For arithmetic operators, the conventional precedences apply. Thus i+j*k is equivalent to i+(j*k). A complete list of infix operator precedences is given in Appendix A.

Infix operators also have associativity, which determines for two consecutive operators of the same precedence, which one applies to which operand. Most operators associate to the left. For example i-j-k is equivalent to (i-j)-k. Exponentiation, however, associates to the right. Thus i**j**k is equivalent to i**(j**k). A complete list of infix operator associativities is given in Appendix A.

2.7 Values and Signals

As indicated in Section 1.1, the result of the evaluation of an expression is both a value and a signal. The value serves the traditional computational role. The signal, success or failure, indicates whether or not a computation completed successfully or whether or not a specified relation held. For example, i = j succeeds if i is equal to j and fails otherwise. The value returned on success is the value of j.

When an expression fails, the value is not used and the failure is passed on to any larger expression of which it is a part. For example

```
i < j < k
```

is equivalent to

```
(i < j) < k
```

This expression fails if i is not less than j or if j is not less than k.

2.8 Control Structures

Expressions ordinarily are evaluated in the sequence in which they appear in the program. Various control structures provide for other orders of evaluation.

2.8.1 Basic Control Structures

Icon contains a number of traditional control structures. These control structures are driven by signals (rather than by boolean values as in most programming languages).

1. The control structure

   if *expr1* **then** *expr2* [**else** *expr3*]

evaluates *expr1*. If *expr1* succeeds, *expr2* is evaluated; otherwise *expr3* is evaluated. The result returned by **if-then-else** is the result of *expr2* or *expr3*, whichever is evaluated. If the **else** clause is omitted and *expr1* fails, the result of the **if-then-else** expression is &null and the signal success.

2. The control structure

   while *expr1* **do** *expr2*

evaluates *expr1* repeatedly until *expr1* fails. Each time *expr1* succeeds, *expr2* is evaluated.

3. The control structure

   until *expr1* **do** *expr2*

evaluates *expr1* repeatedly until *expr1* succeeds. Each time *expr1* fails, *expr2* is evaluated.

Some control structures are designed specifically to use the signaling mechanism of control.

4. The control structure

repeat expr

evaluates expr repeatedly until expr fails.

Note: **repeat** succeeds and returns &null when expr fails.

Note: **while-do**, **until-do**, and **repeat** all return &null and the signal success on completion, regardless of the success or failure of expressions within them.

5. The control structure

expr **fails**

succeeds if expr fails and fails if expr succeeds. For example,

if expr1 **fails then** expr2 **else** expr3

is equivalent to

if expr1 **then** expr3 **else** expr2

Note: **fails** returns &null when it succeeds.

6. The control structure

null expr

succeeds if the value of expr is &null.

Note: If the value of expr is of type other than &null, an attempt is made to convert the value to &null.

examples:

	value	signal
null ""	&null	success
null 0	&null	success
null 0.0	&null	success
null "0"		failure

2.8.2 Compound Expressions

Expressions may be compounded to allow several expressions to appear in a control structure that specifies only a single expression. A compound expression has the form

{ [expr [; expr] ...]}

For example

```
if z = 0 then {x := 0; y := 0}
```

sets both x and y to zero if z is zero.

If the expressions of a compound expression are placed on separate lines, the semicolons are not necessary. For example,

```
if z = 0 then {
   x := 0
   y := 0
}
```

is equivalent to the line above. See also Section 9.1.1.

The result of a compound expression is the signal and value of the last expression in the sequence.

2.8.3 Generators

One of the unusual aspects of Icon is the concept of generators. Some expressions are capable of generating a series of values to obtain successful evaluation of the expression in which they occur.

The most fundamental generator is alternation

expr1 | *expr2*

which first evaluates *expr1*. If *expr1* succeeds, its value is returned. If *expr1* fails, however, *expr2* is evaluated and its result (value and signal) is returned by the evaluation. For example,

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

succeeds if i is equal to j or if j is equal to k.

Alternation has an important additional property. If *expr1* is successful, but the expression in which the alternation occurs would fail, the alternation operator then evaluates *expr2*. For example

```
x = (1 | 3)
```

succeeds if x is equal to 1 or 3.

Another generator is

expr1 **to** *expr2* [**by** *expr3*]

which generates, as required, the integers from *expr1* to *expr2* inclusive, using *expr3* as an increment. For example

```
x = (0 to 10 by 2)
```

succeeds if x is equal to any of the even integers between 0 and 10, inclusive.

Notes: expr3 is evaluated only once. Generation continues until expr2 is reached or exceeded. expr3 may be negative, in which case successively smaller values are generated until expr2 is reached.

Default: If the **by** clause is omitted, the increment defaults to 1.

2.8.4 Goal-Directed Evaluation

Goal-directed evaluation, in which a successful result is sought, is implicit in the examples of the previous section. If a component of an expression fails, evaluation is continued until all alternatives have been attempted.

There are two control structures that are expressly concerned with goal-directed evaluation.

1. The control structure

```
every expr1 [do expr2]
```

produces all alternatives of expr1. For each alternative that is generated, expr2 is evaluated. For example,

```
every i := (1 | 4 | 6) do f(i)
```

calls f(1), f(4), and f(6). Similarly,

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

calls f(1), f(2), ..., f(10).

The **every** loop continues until all alternatives in expr1 have been generated. **every-do** succeeds and returns &null when it completes.

Note: **every** i := j **to** k **do** expr is equivalent to the for control structure found in many programming languages.

2. The conjunction operator

```
expr1 & expr2
```

succeeds only if both expr1 and expr2 succeed. For example

```
(x = y) & (z = 1)
```

succeeds only if x equals y and z equals 1.

If expr1 succeeds but expr2 would fail, alternatives in expr1 are sought in an attempt to obtain successful evaluation of the entire expression. For example

```
(x := 1 to 10) & (x > y)
```

succeeds and assigns to x the least value between 1 and 10 that is greater than y, provided such a value exists.

Note: The conjunction operator does not perform any backtracking. If it fails because expr2 fails, the results of intermediate evaluations are not reversed. In the example above, if the value of y were 20, the value of x after failure of the conjunction would be 10.

2.8.5 Loop Exits

There are two control structures for bypassing the normal completion of expressions in loops. These control structures may be used in **repeat** and in the **do** clauses of **every**, **until**, and **while**.

1. The control structure **next** causes immediate transfer to the beginning of the loop without completion of the expression in which the **next** appears.

2. The control structure **break** causes immediate termination of the loop without the completion of the expression in which the **break** appears.

2.9 Procedures

A program is composed of a sequence of procedures. Procedures have the form

```
procedure name ( argument-list )
   procedure-body
end
```

The procedure name identifies the procedure in the same way that built-in functions are named. The argument list consists of the identifiers through which values are passed to the procedure. The procedure body consists of a sequence of expressions that are evaluated when the procedure is invoked. A **return** expression terminates an invocation of the procedure and returns a value.

An example of a procedure is

```
procedure max(i,j)
   if i > j then return i else return j
end
```

A procedure is invoked in the same fashion that a built-in function is called. For example

```
m := max(size(s1),size(s2))
```

assigns to m the maximum of the sizes of s1 and s2.

Program execution begins with an invocation of the procedure named main. All programs must have a procedure with this name.

For a more detailed description of procedures, see Chapter 8.

CHAPTER 3

Arithmetic Operations

Icon provides integer, real, and mixed-mode arithmetic with the standard operations and comparisons.

3.1 Integers

Integers in Icon are treated as in most programming languages. The allowable range of integer values is machine dependent.

Note: For machines that perform arithmetic in two's-complement form, the absolute value of the largest negative integer is one greater than the largest positive integer.

3.1.1 Literal Integers

Integers may be specified literally in a program in the standard fashion.

Notes: Leading zeroes are allowed but are ignored. Negative integers cannot be expressed literally, but may be computed as the result of arithmetic operations.

examples:

	value
0	0
000	0
10	10
010	10
27524	27524

Integer literals such as those given above are in the base 10. Other radices may be specified by beginning the integer literal with nr, where n is a number (base 10) between 2 and 36 that specifies the radix for the digits that follow. For digits with a decimal value greater than 9, the letters a, b, c, ... are used.

Note: The digits used in the literal must be less than the radix.

examples:

	decimal value
2r11	3
8r10	8
10r10	10
16rff	255
36rcat	15941

3.1.2 Integer Arithmetic

The following infix arithmetic operations are provided.

		relative precedence	associativity
i + j	addition	1	left
i - j	subtraction	1	left
i * j	multiplication	2	left
i / j	division	2	left
i ** j	exponentiation	3	right

Note: The remainder of integer division is discarded, that is, the result is truncated.

Error Conditions: If the result of an arithmetic operation exceeds the range of allowable integer values, Error 203 occurs. On some computers, exceeding arithmetic limits may cause abnormal program termination. If an attempt is made to divide by 0, Error 201 occurs.

examples:

	value
1 + 2	3
1 - 2	-1
1 * 2	2
1 / 2	0
2 / 1	2
2 ** 3	8
2 ** 0	1
2 ** -1	0
1 - 1 - 1	-1
1 * 2 / 2	1
1 / 2 * 2	0
2 / 2 - 1	0
2 / (1 - 2)	-2
4 ** 3 ** 2	262144

The function mod(i,j) produces the residue of i mod j, that is, the remainder of i divided by j. The sign of the result is the sign of i.

Error Condition: If j is 0, Error 202 occurs.

examples:

	value
mod(4,3)	1
mod(4,4)	0
mod(-4,3)	-1
mod(4,-3)	1
mod(-4,-3)	-1

The two prefix operators +i and -i are equivalent to 0 + i and 0 - i, respectively. That is, -i is the negative of i.

examples:

	value
+1	1
-1	-1
-0	0
+0	0
-(4 - 7)	3

There are also two suffix operators that apply to variables that have integer values.

1. The operation i+ increments the value of i by 1 and produces the new value.

2. The operation i- decrements the value of i by 1 and produces the new value.

Error Condition: If the operand of i+ or i- is not a variable, Error 121 occurs.

examples:

expressions in sequence	value of i
i := 1	1
i+	2
i+++	5
i-	4
i----	0

3.1.3 Integer Comparison

There are six operations for comparing the magnitude of integers.

i = j	equal to
i ~= j	not equal to
i > j	greater than

i >= j	greater than or equal to
i < j	less than
i <= j	less than or equal to

All the comparison operators associate to the left and have lower precedence than any of the arithmetic computation operations.

The operations succeed if the specified relation between the operands holds and fail otherwise. The value returned on success is j.

examples:

	value	signal
1 = 1	1	success
1 ~= 1		failure
1 > 1		failure
2 > 1	1	success
1 < 2	2	success
2 >= 1	1	success
2 <= 2	2	success
2 < 3 < 4	4	success
2 < 3 = 4		failure

3.2 Real Arithmetic

Real numbers are represented in floating-point format. The range and precision of real numbers is machine dependent.

3.2.1 Literal Real Numbers

Real numbers may be specified literally in a program in the standard fashions using either decimal or exponent notation.

Note: For magnitudes less than 1, a leading zero is required. Additional leading zeroes are allowed but are ignored.

examples:

	value
3.14159	3.14159
0.0	0.0
000.	0.0
27e2	2700.0
27e-6	0.000027
27e5	2700000.0

3.2.2 Real Arithmetic

The same arithmetic operations are available for real numbers as are available for integers. See Section 3.1.2.

Note: Some systems do not support exponentiation of real numbers.

Error Condition: If an attempt is made to raise a negative real number to a real power, Error 206 occurs.

examples:

	value
1.0 + 2.0	3.0
1.0 - 2.0	-1.0
1.0 * 2.0	2.0
1.0 / 2.0	0.5
2.0 / 1.0	2.0
1.0 -1.0 - 1.0	-1.0
1.0 * 2.0 / 2.0	1.0
1.0 / 2.0 * 2.0	1.0
mod(4.7,2.0)	0.7
mod(2.5,1.0)	0.5

expressions in sequence	value of x
x := 3.1416	3.1416
x+	4.1416
x+++	7.1416
x-	6.1416

3.2.3 Comparison of Real Numbers

The same comparison operations are available for real numbers as are available for integers. See Section 3.1.3.

Note: Because of the imprecision of the floating-point representation and computation, comparison for equality of real numbers may not always produce the result that would be obtained if true real arithmetic were possible.

examples:

	value	signal
1.0 = 1.0	1.0	success
1.0 ~= 1.0		failure
1.0 > 1.0		failure
2.0 > 1.0	1.0	success
1.0 < 2.0	2.0	success
2.0 <= 1.0		failure
2.0 <= 2.0	2.0	success
2.0 < 3.0 < 4.0	4.0	success
2.0 < 3.0 <= 4.0	4.0	success
2.0 < 3.0 = 4.0		failure

3.3 Mixed-Mode Arithmetic

Except for exponentiation, if either operand of an infix operation is real, the other operand is converted to real and real arithmetic is performed. In the case of exponentiation, a negative real number may be raised to an integer power.

examples:

	value
1.0 + 2	3.0
1 + 2.0	3.0
1 - 2.0	-1.0
1.0 * 2	2.0
1.0 / 2	0.5
2 / 1.0	2.0
1 - 1 - 1.0	-1.0
1 * 2.0 / 2	1.0
1 / 2.0 * 2	1.0
1.0 / 2 * 2	1.0
2.0 ** 2	4.0
2.0 ** -1	0.5

3.4 Arithmetic Type Conversion

3.4.1 Conversion to Integer

The value of integer(x) is an integer corresponding to x, where x may have type **integer**, **real**, **string**, **cset**, or **null**.

Failure Condition: integer(x) fails if the type of x is not one of those listed above.

1. An object of type **integer** is returned unmodified by integer(x)

2. An object of type **real** is converted to integer by truncation.

Failure Condition: Conversion of a real to integer fails if the value of the real number is out of the allowable range of integers.

examples:

	value	signal
integer(2.0)	2	success
integer(2.5)	2	success
integer(-2.5)	-2	success
integer(2e75)		failure

3. For type **string**, the string is converted to integer in the same way that an integer literal is treated in program text, except that leading and trailing blanks are allowed, but are ignored. See Section 3.1.1.

Notes: An initial sign is allowed in conversion of a string to an integer. If the string corresponds to a real literal, real-to-integer conversion is performed. See Section 3.4.2. The null string is converted to the integer 0.

Failure Condition: integer(s) fails if s is not a proper representation of an integer or real.

examples:

	value	signal
integer("10")	10	success
integer("8r10")	8	success
integer("-10")	-10	success
integer(" 3")	3	success
integer(" 0003")	3	success
integer("3.5")	3	success
integer("")	0	success
integer("3.x")		failure
integer("3r4")		failure

4. Objects of type **cset** are converted to string and then to integer. See Section 4.4.

5. For type **null**, the value of integer(x) is 0.

For operations that require objects of type **integer**, implicit conversions are automatically performed for the types **real**, **string**, **cset**, and **null**.

Error Condition: If conversion fails, Error 101 occurs.

examples:

	value
1 + "10"	11
2 ** 3.7	8
1 > &null	0

3.4.2 Conversion to Real

The value of real(x) is a real number corresponding to x, where x may have type **real**, **integer**, **string**, **cset**, or **null**.

Failure Condition: real(x) fails if the type of x is not one of those listed above.

1. An object of type **real** is returned unmodified by real(x).

2. An object of type **integer** is converted to the corresponding real value.

examples:

	value
real(10)	10.0
real(-10)	-10.0
real(8r10)	8.0
real(2700000)	2.7e6

3. For type **string**, the string is converted to a real number in the same way as a real literal is treated in program text, except that

(1) Leading and trailing blanks are allowed, but are ignored. See Section 3.2.1.

(2) A leading sign may be included.

(3) A leading zero is not required before the decimal point for values whose magnitudes are less than 1.

Notes: If the string corresponds to an integer literal, integer-to-real conversion is performed. See Section 3.4.1. The null string is converted to 0.0.

Failure Condition: real(s) fails if s is not a proper representation of a real or integer.

examples:

	value	signal
real("10.0")	10.0	success
real("-10.0")	-10.0	success
real("2700000")	2.7e6	success
real(" 3.0")	3.0	success
real(" 0003.0")	3.0	success
real("8r10")	8.0	success
real("")	0.0	success
real("3.x")		failure
real("3r4")		failure

4. Objects of type **cset** are first converted to **string** and then to **real**. See Section 4.4.

5. For type **null**, the value of real(x) is 0.0.

For operations that require objects of type **real**, implicit conversions are automatically performed for the types **integer**, **string**, **cset**, and **null**.

Error Condition: If conversion fails, Error 102 occurs.

examples:

	value
1.0 + "10.0"	11.0
"2.0" ** 3	8.0
1.0 > &null	0.0

3.5 Numeric Test

The function numeric(x) succeeds if x is of type **integer**, **real**, or if it is convertible to one of these types. See Section 3.4. The function fails otherwise. If it succeeds, the value returned is the integer or real corresponding to x.

Note: The value of numeric(&null) is the integer 0.

examples:

	value	signal
numeric(0)	0	success
numeric(0.0)	0.0	success
numeric("0")	0	success
numeric("0.0")	0.0	success
numeric("a")		failure
numeric("3r4")		failure
numeric("")	0	success
numeric(&null)	0	success

CHAPTER 4

String Processing

4.1 Characters

Although characters are not themselves data objects in Icon, strings of characters are, and strings are important in many situations, forming the heart of Icon's processing capabilities.

Icon uses the ASCII character set [3], which is shown in detail in Appendix F. There are 128 characters in the ASCII character set. Some are associated with graphics and are used for representing text and for producing printed output. Other characters have no standard graphics; they typically signify control functions for operating systems and various input and output devices.

Note: The thirty-third character (octal code 40) is the blank (space). Since it has no visible representation the symbol ẟ is used in the body of the text to represent the blank.

Different computer systems use different character sets. For example, the DEC-10 and PDP-11 use ASCII, but the IBM 360/370 uses EBCDIC [5], and CDC 6000 and CYBER systems use both Display Code [6] and ASCII. Despite these differences, the internal character set used by Icon is ASCII and translations are performed upon input and output for computers that use different character sets.

While it is customary to think of characters in terms of their graphic representations and control functions, the 128 ASCII characters are basically just 128 integers in sequence. Internally these integers are represented by octal codes from 000 to 177. The order of characters is determined by these codes and specifies the "collating sequence" of the ASCII character set. For example, Z comes before z in the collating sequence. This order is the basis for comparing strings (see Section 4.6) and for sorting (see Section 5.6).

4.2 Strings

A string is a sequence of zero or more characters. Any character in the ASCII character set may appear in a string. There are many ways of constructing strings during program execution. See Section 4.5.

4.2.1 Literal Strings

Strings may be specified literally in a program by delimiting (enclosing) the sequence of characters by double quotes or single quotes. The same type of quote must be used at the beginning and end of each string literal, and a quote of one type cannot appear directly in a literal delimited by that type (see below).

examples:

literal	string
"X"	X
'X'	X
" "	ßß
"abcd"	abcd
"Isn't the temptation great?"	Isn't the temptation great?
'He yelled "whoopee".'	He yelled "whoopee".

Note: In this manual, string values are given in the body of the text without the delimiting quotation marks.

Some characters cannot be entered directly in program text because of their control functions. To allow specification of all ASCII characters in literal strings, an escape convention is used in which the backslash (\) causes subsequent characters to have a special meaning:

backspace	\b
delete	\d
escape	\e
form feed	\f
line feed	\l
carriage return	\r
horizontal tab	\t
vertical tab	\v
double quote	\"
single quote	\'
backslash	\\
octal code	\ddd

The specification \ddd represents the character with octal code ddd (see Appendix F). Only enough digits need to be given to specify the code. For example, \0 specifies the null character. If the character following a backslash is not one of those listed above, the backslash is ignored.

Notes: The convention used here for representing ASCII characters in literals is adapted from that used by the C programming language [4]. Since |, \, and ! are all equivalent in their syntactic interpretation, any of these characters may be used as the escape character and to be represented literally, all these characters must themselves be preceded by an escape character.

Warning: If |, \, or ! is intended to be used literally, it must be preceded by an escape character. Otherwise unexpected results or a syntactically erroneous construction may occur.

examples:

literal	string
"\\"	\
"\"oops\""	"oops"
"\"\""	""
'\'\''	''
'\ '	ϐ
"\a\x"	ax
"\132"	z
"\134\134"	\\
"!!"	!
"|!"	!
"||"	|
"!\"	\

4.2.2 Built-In Strings

The protected keyword &ascii consists of a string of all the ASCII characters in collating sequence.

Warning: Ordinarily the value of &ascii should not be transmitted to an output device, since some ASCII characters are typically used to control such devices.

The letters of the alphabet are used so frequently that two protected keywords are provided for convenience:

keyword	value
&ucase	ABCDEFGHIJKLMNOPQRSTUVWXYZ
&lcase	abcdefghijklmnopqrstuvwxyz

4.2.3 String Length

The length of a string is the number of characters it contains and is computed by size(s).

examples:

	value
size("abcd")	4
size(&lcase)	26
size(" ")	1
size(&ascii)	128

The maximum length of a literal string (excluding the delimiters

and special encodings) is 120. Strings constructed during program execution are limited in length by internal, machine-dependent considerations. The practical maximum is usually dictated by the amount of memory available. In any event, strings may be very long, although the manipulation of long strings is expensive.

4.2.4 The Null String

The null string is the string consisting of no characters and has length zero. It may be represented literally by two adjacent quotes, enclosing no characters. The null string is also produced by the use of the keyword &null in a context that requires a string.

examples:

	value
size()	0
size("")	0
size(&null)	0

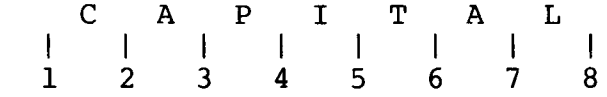
The default initial value of identifiers of type **string** is the null string.

Notes: Since the null string contains no characters, it has no visible representation. In this manual, the symbol ø is used to represent the null string in the body of the text. The null character (see Appendix F) is not related to the null string. A string consisting of a single null character has a length of 1.

4.2.5 Positions of Characters in a String

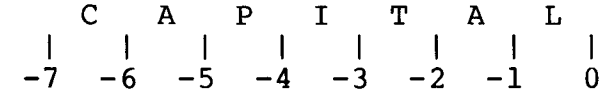
The positions of characters in a string are numbered from the left starting at 1. The numbering identifies positions between characters.

example: The positions in the string CAPITAL are



Note that the position after the last character may be specified.

Positions may also be specified with respect to the right end of a string, using nonpositive numbers starting at 0 and continuing with negative values toward the left:



For this string, positions 8 and 0 are equivalent, positions -1 and 7 are equivalent, and so on.

Note: The only allowable positions for the null string are 1 and 0, which are equivalent.

The positions that can be specified for a string s are in the range -size(s) <= i <= size(s)+1. Other values are out of range and are not allowable position specifications.

In general, the positive specification i is equivalent to the negative specification -(size(s)+1)+i. While nonpositive position specifications are frequently convenient, it is also often necessary to express position specifications in their positive form. The value of pos(i,s) is the positive position specification of i with respect to s, regardless of whether i is in positive form or not.

Failure Condition: pos(i,s) fails if i is out of range of s.

examples:

	value	signal
pos(0,&lcase)	27	success
pos(-1,&lcase)	26	success
pos(1,&lcase)	1	success
pos(28,&lcase)		failure
pos(0,&null)	1	success
pos(-1,&null)		failure

Default: pos(i) defaults to a special meaning for string scanning. See Section 4.8.3.

4.3 Character Sets

Whereas a string is an ordered sequence of characters in which the same character may appear more than once, a character set (type **cset**) is an unordered collection of characters. Character sets are subsets of the ASCII character set and are useful for operations on strings where specific characters are of interest, regardless of the order in which they appear. See Sections 4.7.2 and 4.8.3.

The value of cset(s) is a character set consisting of the characters in s. Duplicate characters in s are ignored, and the order of characters in s is irrelevant.

examples:

	characters
cset("abcd")	a b c d
cset("badc")	a b c d
cset("energy")	e g n r y

The value of ~c is the complement of c with respect to &ascii,

that is, a character set containing all ASCII characters that are not contained in c.

Note: A character set may be empty, i.e. containing no characters. Such a character set may be obtained by cset(&null) or ~cset(&ascii).

4.4 Type Conversions

The value of string(x) is a string corresponding to x, where x may have type **integer**, **real**, **string**, **cset**, **file**, or **null**.

Failure Condition: string(x) fails if the type of x is not one of those listed above.

1. An object of type **string** is returned unmodified by string(x).

2. For the numeric types **integer** and **real**, the resulting string is a representation of the numerical value corresponding to the literal representation that the numeric object would have in the source program.

Note: Literal representations are in normal form -- without leading zeroes and according to the following rules for reals:

(1) Trailing zeroes are suppressed.

(2) The number of significant digits depends on the precision of reals and is machine dependent. In the examples that follow, a precision of five digits is assumed.

(3) If the absolute value of the real number is greater than 10^5 or less than 10^{-4}, the exponent notation is used.

examples:

	value
string(10)	10
string(00010)	10
string(8r10)	8
string(2.7)	2.7
string(02.70)	2.7
string(27e-1)	2.7
string(2700000.)	2.7e6
string(0.0000027)	2.7e-6

3. For type **cset**, the value is a string of characters in the cset, arranged in order of the ASCII character set (see Section 4.6).

Note: Conversion of a string to a cset and back to string, as in

```
s := string(cset(s))
```

eliminates duplicate characters and sorts the characters of the string.

examples:

	value
string(cset("abcd"))	abcd
string(cset("badc"))	abcd
string(cset("energy"))	egnry
string(cset("a + b"))	␣+ab

4. For type **file**, the value of string(x) is the file name. The representation of file names, as well as the normal forms of equivalent files is machine and system dependent. See Section 6.1.

5. For type **null**, the value of string(x) is ø.

There are two other functions that convert objects of various types to strings.

1. The function type(x) returns a string that is the name of the type of x.

2. The function image(x) produces a string that resembles the form the value of x would have in the text of a program. For strings, this includes enclosing quotes and escapes as necessary.

examples:

	value
type(1)	integer
type(2.0)	real
type("")	string
type()	null
image(1)	1
image(2.0)	2.0
image('abc')	"abc"
image("")	""
image()	&null

For operations that require objects of type **string**, implicit conversions are automatically performed for the types **integer**, **real**, **cset**, **file**, and **null**.

Error Condition: If an object of any other type is encountered in a context that requires a string, Error 104 occurs.

examples:

	value
pos(0,10)	3
size(010)	2
size(10)	2
size(&null)	0

Similary, for operations that require objects of type **cset**, implicit conversion is performed automatically for types **integer**, **real**, **string**, and **null**. The conversions are performed by first converting to type **string**, if necessary, and then to type **cset**.

Error Condition: If an object of any other type is encountered in a context that requires a character set, Error 105 occurs.

examples:

	characters
cset(1088)	0 1 8
cset(3.14159)	. 1 3 4 5 9

Note: In this manual, arguments of type **cset** are usually given as strings for clarity.

4.5 Constructing Strings

There are a number of operations for constructing strings. Most of these operations are descibed in the following sections. See also Sections 4.8.2 and 4.8.3.

4.5.1 Concatenation

Since a string is a sequence of characters, one of the most natural string construction operations is concatenation, appending one string to another. The value of s1 || s2 is a string consisting of s1 followed by s2.

Note: The null string is the identity with respect to concatenation. That is, the result of concatenating the null string with any string s is simply s.

examples:

	value
"a" || "z"	az
"[" || "abcd" || "]"	[abcd]
"abcd" || &null	abcd
"" || ""	ø

4.5.2 String Replication

The value of repl(s,i) is the result of concatenating i copies of s.

Error Condition: In repl(s,i), if i is negative, Error 211 occurs.

Note: The value of repl(s,0) is ø.

examples:

	value	signal
repl("a",3)	aaa	success
repl("*.",3)	*.*.*.	success
repl(&lcase)	ø	success

4.5.3 Positioning Strings for Column Output

When text is printed in columns, it is useful to position data in strings of a specified size. There are three functions for doing this.

1. The value of left(s1,i,s2) is s1 positioned at the left of a string of size i. s2 is used to fill out the remaining portion to the right of s1, and is replicated as necessary, starting from the right. The last copy of s2 is truncated at the left if necessary to obtain the proper size. If the size of s1 is greater than i, it is truncated at the right end.

Default: A null or omitted value of s2 defaults to " ".

Error Condition: In left(s1,i,s2), if i is negative, Error 212 occurs.

examples:

	value
left("abcd",6,". ")	abcd.ß
left("abcd",7,". ")	abcdß.ß
left("abcde",7",. ")	abcde.ß
left("abcd",6)	abcdßß
left(&lcase,10)	abcdefghij

2. The value of right(s1,i,s2) is similar to left(s1,i,s2), except that s1 is placed at the right, s2 is replicated starting at the left, with the truncation of the last copy of s2 at the right if necessary.

Default: A null or omitted value of s2 defaults to " ".

Error Condition: In right(s1,i,s2), if i is negative, Error 213 occurs.

examples:

	value
right("abcd",6,". ")	.ßabcd
right("abcd",7,". ")	.ß.abcd
right("abcde,7,". ")	.ßabcde
right("abcd",6)	ßßabcd
right(&lcase,10)	qrstuvwxyz

3. The value of center(s1,i,s2) is s1 centered in a string of length i. s2 is used for filling on the left and right as for the functions above. If s1 cannot be centered exactly, it is placed one character to the left of center.

Default: A null or omitted value of s2 defaults to " ".

Error Condition: In center(s1,i,s2), if i is negative, Error 214 occurs.

examples:

	value
center("abcd",8,". ")	.ßabcd.ß
center("abcd",9,". ")	.ßabcdß.ß
center("abcde",9,". ")	.ßabcde.ß
center("abcd",6)	ßabcdß
center(&lcase,10)	ijklmnopqr
center(&lcase,11)	hijklmnopqr

4.5.4 Substrings

A substring is a sequence of characters within a string. An initial substring of s is one that begins at the first character of s. A terminal substring of s is one that ends at the last character of s. There are four operations that return substrings.

1. The value of section(s,i,j) is the substring of s between positions i and j, inclusive.

Failure Conditions: section(s,i,j) fails if i or j is out of range.

Default: section(s) defaults to section(s,1,0), i.e., the entire string.

examples:

	value	signal
section(&lcase,1,2)	a	success
section(&lcase,2,1)	a	success
section(&lcase,1,1)	ø	success
section(&lcase,27,28)		failure
section(&lcase,-1,-2)	y	success
section("abcd",2)	bcd	success
section("abcd",2,-7)		failure
section("abcd")	abcd	success

If the first argument of section is a variable, assignment to section(s,i,j) can be performed to replace the specified substring and hence change the value of s.

examples:

expressions in sequence	value of s
s := "abcd"	abcd
section(s,1,2) := "xx"	xxbcd
section(s,-1,0) := ""	xxbc
section(s,1,1) := "abc"	abcxxbc

2. The value of substr(s,i,j) is the substring of length j starting at position i of s. The length specification may be negative, indicating a string taken from i toward the left.

Note: substr(s,i,j) is equivalent to section(s,i,j+pos(i,s)).

Failure Conditions: substr(s,i,j) fails if i or pos(i)+j is out of range.

examples:

	value	signal
substr(&lcase,2,1)	b	success
substr(&lcase,2,26)		failure
substr("abcd",2,2)	bc	success
substr(&lcase,-1,-2)	xy	success
substr(&lcase,,-1)	z	success
substr("abcd",2,0)	ø	success
substr("abcd",2)	ø	success
substr("abcd",2,-3)		failure

Assignment to substr(s,i,j) can be performed in the same manner as to section(s,i,j).

examples:

expressions in sequence	value of s
s := "abcd"	abcd
substr(s,1,1) := "x"	xbcd
substr(s,2,-1) := ""	bcd
substr(s,1) := "xxx"	xxxbcd

3. The expression s[i] is equivalent to substr(s,i,1).

examples:

	value	signal
&lcase[1]	a	success
&lcase[0]		failure
&ascii[98]	a	success
&ascii[33]	ß	success
&lcase[-1]	z	success
"abcd"[-2]	c	success
&null[2]		failure

Warning: The internal representation of characters starts at 0, not 1, while the positions in a string start at 1. Consequently, there is a difference of 1 between the position of a character in &ascii and its (decimal) code value (see Appendix F). Thus &ascii[1] is the null character. This difference may be an annoyance and also a source of error. It is the consequence of the technique used for specifying positions from either end of the string by unique integers.

Assignment to s[i] can be performed in the same manner as to substr(s,i,1). For example

```
s[3] := "xy"
```

replaces the third character of s by xy. Similarly,

```
s[3] :=: s[4]
```

exchanges the third and fourth characters of s.

examples:

expressions in sequence	value of s
s := "abcd"	abcd
s[2] := "x"	axcd
s[1] := s[-1]	dxcd
s[2] :=: s[3]	dcxd
s[1] := "abcd"	abcdxcd
s[1] := &null	bcdxcd

4. The operation !s is a generator that produces, as required, s[1], s[2], ..., s[size(s)].

Note: Assignment to !s may be performed in the same manner as to s[i].

examples:

	sequence of values
every !"abcde"	a, b, c, d, e
every !section(&lcase,10,15)	j, k, l, m, n, o

4.5.5 Other String-Valued Operations

1. The value of reverse(s) is a string consisting of the characters of s in reverse order.

examples:

	value
reverse("abcd")	dcba
reverse(&lcase)	zyxwvutsrqponmlkjihgfedcba
reverse(&null)	ø

2. The value of trim(s,c) is a string consisting of the initial substring of s with the omission of the trailing substring of s which consists solely of characters contained in c.

Default: trim(s) defaults to trim(s,cset(" ")).

examples:

	value
trim("abcd "," ")	abcd
trim("abcd ")	abcd
trim("abcd "," d")	abc
trim("abcd ","d")	abcdßßß
trim("abcd ",&ascii)	ø

3. The value of map(s1,s2,s3) is a string resulting from a character mapping on s1, where each character of s1 that is contained in s2 is replaced by the character in the corresponding position in s3. Characters of s1 that do not appear in s2 are left unchanged. If the same character appears more than once in s2, the rightmost correspondence with s3 applies.

Error Condition: If the lengths of s2 and s3 are not the same, Error 215 occurs.

Note: If s1 is a transposition (rearrangement) of the characters of s2, then map(s1,s2,s3) produces the corresponding transposition of s3.

examples:

	value
map("abcda","a","*")	*bcd*
map("abcda","ad","**")	*bc**
map("abcda","ad","*:")	*bc:*
map("abcda","ax","*:")	*bcd*
map("abcda","yx","*:")	abcda
map("abcda","bcad","1234")	31243
map("abcda","abac","1234")	324d3
map("wxyz","zyxw","abcd")	dcba

4.6 String Comparison

Strings, like numbers, can be compared, but the basis for comparison is lexical (alphabetical) order rather than numerical value. Lexical order includes all characters of the ASCII character set and is based on the collating sequence as given by &ascii. If a character c1 appears before c2 in &ascii, c1 is lexically less than c2. The lexical order for single-character strings is based on this ordering. Thus X is less than x, but z is greater than x. For longer strings, lexical order is determined by the lexical order of characters in corresponding positions, starting at the left. Two strings are lexically equal if and only if they are identical, character by character. If one string is an initial substring of another, then the shorter string is lexically less than the longer one.

Note: The null string is lexically less than any other string.

The function llt(s1,s2) succeeds if s1 is lexically less than s2 and fails otherwise. The value returned on success is s2.

examples:

	value	signal
llt("X","x")	x	success
llt("x","X")		failure
llt("x","x")		failure
llt("XX","x")	x	success
llt("xx","xX")		failure
llt("xx","xxx")	xxx	success
llt("xx","xxX")	xxX	success
llt(&null,"x")	x	success
llt(&null,&null)		failure

There are four lexical comparison predicates and two lexical comparison operators:

llt(s1,s2)	lexically less than
lle(s1,s2)	lexically less than or equal
lgt(s1,s2)	lexically greater than
lge(s1,s2)	lexically greater than or equal
s1 == s2	lexically equal
s1 ~== s2	lexically not equal

4.7 String Analysis

The majority of programming operations on strings involve analysis rather than synthesis, and the repertoire of analytic operations is correspondingly large. A higher-level form of string analysis is included in string scanning, which is described in Section 4.8.

4.7.1 Identifying Substrings

There are two functions for identifying specific substrings.

1. The function match(s1,s2,i,j) succeeds if s1 is an initial substring of section(s2,i,j). The value returned is the position of the end of the substring, that is, i+size(s1).

Failure Condition: match(s1,s2,i,j) fails if s1 does not exist at the beginning of section(s2,i,j).

Default: Since section(s) defaults to section(s,1,0), match(s1,s2) defaults to match(s1,s2,1,0).

examples:

	value	signal
match("a","abcd",1)	2	success
match("a","abcd")	2	success
match("a","abcd",2)		failure
match("ab","abcd",1,2)		failure
match("bc","abcd",1)		failure
match("bc","abcd",2)	4	success
match("bcde","abcd",2)		failure
match(&null,"abcd",1)	1	success
match(&null,"abcd",5)	5	success

2. The function find(s1,s2,i,j) succeeds if s1 is a substring anywhere in section(s2,i,j). The value returned is the position in s2 where the substring begins.

Failure Condition: find(s1,s2,i,j) fails if s1 does not exist in section(s2,i,j).

Default: find(s1,s2) defaults to find(s1,s2,1,0). With the standard default, an omitted third argument is equivalent to the end of the string.

examples:

	value	signal
find("a","abcd",1)	1	success
find("a","abcd")	1	success
find("bc","abcd",1)	2	success
find("a","abcd",2)		failure
find("ab","abcd",1,2)		failure
find("de","abcd",1)		failure
find(&null,"abcd",3)	3	success

find is a generator that produces, as required, a sequence of the positions, from left to right, at which s1 is a substring of section(s2,i,j).

examples:

	sequence of values
every find("a","abaaa")	1, 3, 4, 5
every find("abcd","abcdeabc")	1
every find("bc","abcdeabc")	2, 7
every find("bc","abcdeabc",3)	7

4.7.2 Lexical Analysis

Lexical analysis operations involve sets of characters rather than substrings. There are four lexical analysis operations.

1. The value of any(c,s,i,j) is i+1 if the first character of section(s,i,j) is contained in the character set c.

Failure Condition: any(c,s,i,j) fails if s[i] is not contained in c.

Default: any(c,s) defaults to any(c,s,1,0).

examples:

	value	signal
any("abc","abcd",1)	2	success
any("abc","abcd")	2	success
any("abc","dcba")		failure
any(~"abc","dcba")	2	success
any("abc","dcba",2)	3	success
any("abcd","abcd",1,1)		failure

2. The value of upto(c,s,i,j) is the position in s of the first instance of a character of c in section(s,i,j).

Failure Condition: upto(c,s,i,j) fails if no character in section(s,i,j) is contained in c.

Default: upto(c,s) defaults to upto(c,s,1,0).

examples:

	value	signal
upto("a","abcd",1)	1	success
upto("a","abcd")	1	success
upto("abc","abcd")	1	success
upto(~"abc","abcd")	4	success
upto("d","abcd",2)	4	success
upto("a","abcd",2)		failure

upto is a generator that produces, as required, a sequence of the positions, from left to right, at which a character of c occurs in section(s,i,j).

examples:

	sequence of values
every upto("abcd","abcd")	1, 2, 3, 4
every upto("a","abcd")	1
every upto("ab","abcd",2)	2
every upto(~"ab","abcd")	3, 4

3. The value of many(c,s,i,j) is the position in s after the longest initial substring of section(s,i,j) consisting solely of characters contained in c.

Failure Condition: many(c,s,i,j) fails if s[i] is not contained in c.

Default: many(c,s) defaults to many(c,s,1,0).

examples:

	value	signal
many("ab","abcd",1)	3	success
many("ab","abcd")	3	success
many("ab","abcd",2)	3	success
many("ab","abcd",3)		failure

4. The value of bal(c1,c2,c3,s,i,j) is the position in s after an initial substring of section(s,i,j) that is balanced with respect to characters in c2 and c3, respectively, and which ends at the end of section(s,i,j) or is followed by a character in c1.

In determining balance, a count is kept starting at 0, as

characters in section(s,i,j) are processed from left to right. A character in c2 causes the count to be incremented by 1, while a character in c3 causes the count to be decremented by 1. All other characters leave the count unchanged. If the count is 0 after processing a character and the termination condition is satisified, the process is complete at that position. Otherwise, it is continued.

Failure Condition: If the count ever becomes negative or if the string is exhausted with a positive count, bal fails.

Notes: Characters in c2 are examined before characters in c3, so if a character occurs in both c2 and c3, it is treated as if it occurred only in c2. By the algorithm described above, at least one character is always processed. If that character is not contained in c2 or c3, the value returned is i+1, provided the termination condition is satisified.

Defaults: bal(c1,c2,c3,s) defaults to bal(c1,c2,c3,s,1,0). If c1 is omitted, it defaults to cset(&ascii). An omitted value of c2 defaults to cset("(") and an omitted value of c2 defaults to cset(")").

examples:

	value	signal
bal("+","(",")","(A)+(B)",1)	4	success
bal("+",,,"(A)+(B)",1)	4	success
bal("+",,,"(A)+(B)")	4	success
bal("-",,,"(A)+(B)")	8	success
bal(,,,"(A)+(B)")	4	success
bal(,"([","])","(A)+(B)")	4	success
bal(,"([","])","[A)+(B]")	4	success
bal(,,"]","[A)+(B]")	2	success
bal(,,,")A(+)B(")		failure

bal is a generator that produces, as required, a sequence of positions, from left to right, at which successively longer balanced strings terminate.

examples:

	sequence of values
every bal(,,,"(A)+(B)+(C)")	4, 5, 8, 9, 12
every bal("+",,,"(A)+(B)+(C)")	4, 8, 12
every bal(,,,"abcd")	2, 3, 4, 5

4.8 String Scanning

String scanning is a high-level facility for the analysis and synthesis of strings that permits the string being operated on to be implicit, thus avoiding much of the notational detail that would otherwise be required.

4.8.1 Scanning Keywords

The string being scanned is the value of the unprotected keyword &subject. The implicit position in &subject is the value of the unprotected keyword &pos. Assignment of a value to &subject automatically sets &pos to 1, the beginning of &subject. &pos may be subsequently changed as desired.

As an example of implicit arguments, find(s) defaults to find(s,&subject,&pos,0). Thus, if many operations of this type are to be performed on the same string, this string can be assigned to &subject and the operations can be written in an abbreviated form.

4.8.2 Positional Synthesis

There are two functions that change &pos automatically and return the substring between the previous and new values of &pos.

The value of move(i) is the substring between &pos and &pos + i, and i is added to &pos.

Failure Condition: If &pos+i is out of range, move(i) fails. and &pos is not changed.

examples:

expressions in sequence	value	signal	&pos
&subject := "abcd"	abcd	success	1
move(2)	ab	success	3
move(3)		failure	3
move(-1)	b	success	2
move(-2)		failure	2
move(0)	ø	success	2

The assignment made to &pos by move(i) is a reversible effect. If move(i) succeeds but the expression in which it appears fails, &pos is restored to its original value.

examples:

expressions in sequence	value	signal	&pos
&subject := "abcd"	abcd	success	1
move(2) & move(3)		failure	1
move(2)	ab	success	3
move(-1) & (&pos = 3)		failure	3

The value of tab(i) is the substring between &pos and i, and &pos is set to i.

Failure Condition: If i is out of range, tab(i) fails and &pos is not changed.

examples:

expressions in sequence	value	signal	&pos
&subject := "abcd"	abcd	success	1
tab(2)	a	success	2
tab(0)	bcd	success	5
tab(1)	abcd	success	1
tab(-5)		failure	1

The assignment made to &pos by tab(i) is a reversible effect.

examples:

expressions in sequence	value	signal	&pos
&subject := "abcd"	abcd	success	1
tab(0) & move(1)		failure	1
tab(0) & move(-1)	d	success	4

4.8.3 Scanning Operations

Several functions have defaults that provide implicit arguments for string scanning. For example, pos(i) defaults to pos(i,&subject). Thus, pos(0) = &pos succeeds if &pos is positioned after the end of &subject. Other defaults are:

form	interpretation
any(c)	any(c,&subject,&pos,0)
bal(c1,c2,c3)	bal(c1,c2,c3,&subject,&pos,0)
find(s)	find(s,&subject,&pos,0)
match(s)	match(s,&subject,&pos,0)
many(c)	many(c,&subject,&pos,0)
upto(c)	upto(c,&subject,&pos,0)

Thus in each case the operation applies to &subject starting at

&pos and continuing to the end of &subject. The values returned by these functions are integers representing positions in &subject, but &pos is not changed.

examples:

expressions in sequence	value	signal	&pos
&subject := "abcd"	abcd	success	1
upto("c")	3	success	1
upto("a")	1	success	1
many("abc")	4	success	1
any("d")		failure	1

These functions may be used as arguments to tab to change the value of &pos and obtain a substring between the new and old values of &pos.

examples:

expressions in sequence	value	signal	&pos
&subject := "abcd"	abcd	success	1
tab(upto("c"))	ab	success	3
tab(upto("a"))		failure	3
tab(many("c"))	c	success	4
tab(any("d"))	d	success	5

In addition, =s is provided as a synonym for tab(match(s)).

examples:

expressions in sequence	value	signal	&pos
&subject := "abcd"	abcd	success	1
="ab"	ab	success	3
="ab"		failure	3
="c"	c	success	4
="d"	d	success	5
=&null	ø	success	5
="d"		failure	5

4.8.4 The Scope of Scanning

Assignment to &subject establishes a global string as the implicit argument to scanning operations. A local scope for scanning is provided by the operation s ? x, which assigns s to &subject and then evaluates x. The result of s ? x is the result of x, where x may be any expression.

examples:

	value	signal
`"abcd" ? tab(upto("d"))`	abc	success
`"abcd" ? {&pos := 0; tab(2)}`	bcd	success
`"abcd" ? move(5)`		failure

The values of &subject and &pos are only changed during the evaluation of s ? x. In fact, s ? x is conceptually equivalent to

```
save(&pos)
save(&subject)
&subject := s
x
restore(&subject)
restore(&pos)
```

where save(v) and restore(v) represent internal operations that temporarily save and restore values.

CHAPTER 5

Structures

Structures are aggregates of variables. Different kinds of structures have different organizations and different methods for accessing these variables. The different organizations and access methods are chosen for their suitability in various programming contexts. Variables in arrays are organized like points in rectangular coordinate systems and are accessed by position, while tables are sets that are accessed by content. Stacks provide last-in, first-out access. Records provide an organization in which fields are accessed by name.

5.1 Arrays

Arrays are accessed by position, using indices that correspond to spatial coordinates. There is a variable, referred to as an array element, for each coordinate position in the array. Arrays may be multidimensional and the origin of the array may be specified.

5.1.1 Creation of Arrays

Arrays are created during program execution by expressions of the form

array [*array-prototype*] [*type*] [*initial-clause*]

The array prototype describes the structure of the array: the number of dimensions, their extent, and the origin of the array. The prototype consists of a list of range specifications, one for each dimension:

range [, *range*] ...

Each range specification describes a dimension, with a lower bound (the origin in that dimension) and an upper bound separated by a colon:

[*lower-bound* :] *upper-bound*

If the lower bound is one, only the upper bound need be given. For example, 10 and 1:10 are equivalent range specifications. The extent of a dimension is the difference between its upper and lower bounds plus one. The size of an array is the product of the extents of its dimensions. Range specifications may be arbitrary expressions, allowing the creation of arrays with computed ranges.

Error Condition: If a range specification is erroneous, Error 216 occurs.

Note: Although there is no limit on the range of a dimension or on the number of dimensions, there is a limit on the size of data objects, which is imposed by machine architecture and the amount of available memory. Such limits are machine and environment dependent.

examples:

	dimensionality	size	origin
list := **array** 1:10	1	10	1
list := **array** 10	1	10	1
board := **array** size(list),size(list)	2	100	1, 1
sector := **array** -5:2	1	8	-5
slab := **array** -5:2,10	2	80	-5, 1
cube := **array** 5,5,5	3	125	1, 1, 1

If an array is typed **any**, it is heterogeneous and its elements may be of different types. If an array has any other type, all elements are of that type.

The initial clause specifies a value that is assigned to all array elements when the array is created.

examples:

```
list := array 10 integer initial 1
bar := array 10 any initial 0
board := array 10,10 string initial "x"
sector := array -5:2 string initial "left"
```

Defaults: As indicated above, all components of the **array** expression are optional. The defaults for omitted components are:

array-prototype	0
type	**any**
initial-clause	**initial** &null

An array prototype of 0 specifies a one-dimensional array with no elements. See Section 5.1.3.

Linear arrays with an origin of 1 are called lists. A list of size n may be created as shown above or by an expression of the form

```
<x1,x2,...,xn>
```

where x1, x2, ..., xn are the initial values of the n elements.

examples:

	size
triple := <0,0,0>	3
line := <,,,>	4
octave := <1,2,3,4,5,6,7,8>	8
unit := <>	1

5.1.2 Accessing Array Elements

An element of an array is accessed by specifying the coordinates in an expression of the form

x[i1,i2,...,in]

where x is an n-dimensional array and i1, i2, ..., in are the coordinates of the element. Coordinates are also called subscripts. Assignment may be made to a reference expression to assign a value to the element.

Note: Omitted or extra expressions in the reference are treated like omitted or extra arguments in function calls. See Section 2.5.

Failure Condition: The referencing expression fails if a coordinate is out of range.

examples:

(for the arrays given in the preceding examples)

	value	signal
list[3]	1	success
list[3] := list[5] * 5	5	success
list[0]		failure
list[]		failure
octave[4]	4	success
unit[1]	ø	success
unit[2]		failure
board[1,1]	x	success
board[12,5]		failure
board[10,0]		failure
board[]		failure
sector[]	left	success

5.1.3 Expandable Arrays

Arrays are ordinarily of fixed size. Lists may be opened for expansion so that they can be indexed beyond the original upper bound. A list is opened by the expression open(list).

Subsequently, the list expands automatically when assignment is made to a reference with a coordinate that is one beyond the current upper bound.

Notes: Expansion occurs only when the coordinate is one beyond the current upper bound. References to larger coordinates fail. Expansion occurs only when an assignment is made; merely a reference to such a coordinate fails.

The open function modifies its argument and also returns its argument as value.

The function close(list) prevents the list from being expanded by out-of-range references.

Default: Lists are ordinarily closed when they are created. A list of size 0, however, is created as an open list.

examples:

expressions in sequence	value	signal	size of list
laundry := **array** 10	array	success	10
laundry[1]	ø	success	10
laundry[11] := "shirts"		failure	10
open(laundry)	array	success	10
laundry[12] := "shirts"		failure	10
laundry[11] := "shirts"	shirts	success	11
laundry[12] := "socks"	socks	success	12
close(laundry)	array	sucesss	12
laundry[12]	socks	success	12
laundry[13]		failure	12

5.2 Tables

A table is an aggregate of elements that resembles a one-dimensional array, A table, however, can be referenced by any object. The elements of a table are not ordered by position. Thus a table can be thought of as an associative array.

5.2.1 Creation of Tables

Tables are created during program execution by expressions of the form

table [size] [ref-type] [, value-type]

When tables are created, they are empty and have no elements. Elements may be added at will (see Section 5.2.3) and tables grow automatically. The size given in the **table** expression limits the number of elements in the table. A size of 0 specifies a table that is not limited in size, except by the availability of memory.

Default: An omitted size defaults to 0.

Error Condition: If a size specification is negative, Error 218 occurs.

5.2.2 Accessing Table Elements

An element of a table is accessed by specifying a referencing value in an expression of the form t[x] where t is a table and x is the referencing value. The referencing value need not be an integer. For example, t["n"] references the table t with the string n.

A value is assigned to a table element in a manner similar to that for arrays. For example

```
t["n"] := 3
```

assigns the integer 3 to the element referenced by the string n.

Defaults: The types for the reference and value of elements of a table are optional. An unspecified type defaults to **any**. Note that a comma separates the two type specifications.

examples:

	type default
table	**table any, any**
table string	**table string, any**
table ,string	**table any, string**

A table grows automatically as assignments are made to referenced elements that are not already in the table.

The value of a table element that is not in the table is &null, and is converted to the expected type. Table elements are only created, however, when values are assigned to them. See also Section 5.2.3.

Error Condition: If an attempt is made to exceed the specified maximum size of a table, Error 301 occurs.

examples:

expressions in sequence	value	size of table
opcode := **table**	table	0
opcode["ADD"] := 273	273	1
opcode["SUB"]	ø	1
opcode["SUB"] := 274	274	2
count := **table string, integer**	table	0
count["four"] := count["four"] + 1	1	1
count["score"] := count["score"] + 1	1	2
count["1776"] := count["1776"] + 1	1	3

count[1776]	1	3
count[1776] := "000"	0	3

5.2.3 Closed Tables

As discussed above, tables are ordinarily expandable and grow as values are assigned to newly referenced elements. Tables may be closed to prevent future expansion. A table is closed by close(t) where t is a table. When a table is closed, new elements cannot be added, but existing elements can be accessed or assigned new values.

Failure Condition: When a table is closed, a reference to a non-existent element fails.

The function open(t) opens t for further expansion.

examples:

expressions in sequence	value	signal
digram := **table string, integer**	table	success
digram["th"] := 73	73	success
digram["en"] := 81	81	success
digram["io"] := 41	41	success
close(digram)	table	success
digram["th"] := digram["th"] + 1	74	success
digram["st"]		failure
open(digram)	table	failure
digram["st"]	0	failure

5.3 Stacks

A stack is an aggregate of variables that resembles a one-dimensional array. A stack, however, grows and shrinks automatically as elements are added (pushed) and deleted (popped). Furthermore, a stack can be accessed only at the most recently added element (top).

5.3.1 Creation of Stacks

Stacks are created during program execution by expressions of the form

stack [size] [type]

The size expression limits the maximum number of elements the stack may have. A size of 0 specifies a stack of unlimited size.

Default: An omitted size defaults to 0.

Error Condition: If a size specification is negative, Error 217 occurs.

The type specifies the types of elements that may be added to the stack.

Default: An omitted type specification defaults to **any**.

5.3.2 Accessing Stacks

When a stack is created, it is empty and contains no elements. An element is added to a stack by the function push(k,x), where k is a stack and x is a value to be added to the top of the stack. The value of push(k,x) is x.

Error Condition: If an attempt is made to exceed the specified maximum size of a stack, Error 302 occurs.

An element is removed from a stack by the function pop(k). The value of pop(k) is the value that is removed.

The top element of a stack is referenced by top(k), which returns the value of the top element of k. Assignment may be made to top(k) to change the value of the top element of the stack.

Failure Condition: pop(k) and top(k) fail if k is empty.

examples:

expressions in sequence	value	signal	size of stack
pstack := **stack**	stack	success	0
push(pstack,"x")	x	success	1
push(pstack,"y")	y	success	2
push(pstack,"*")	*	success	3
top(pstack)	*	success	3
pop(pstack)	*	success	2
top(pstack) := "z"	z	success	2
pop(pstack)	z	success	1
pop(pstack)	x	success	0
pop(pstack)		failure	0
top(pstack)		failure	0

5.4 Records

Records are aggregates of variables that resemble one-dimensional arrays, but the elements are accessed by name rather than position.

5.4.1 Declaring Record Types

A record type is declared in the form

```
record name
   field-1 [:type-1],
   field-2 [:type-2],
           .
           .
           .
   field-n [:type-n]
end
```

Note: A record declaration cannot appear within a procedure declaration or within another record declaration.

The name specifies a new type, which is added to the repertoire of types and becomes a reserved word.

The fields provide names for the n elements of the record and have type specifications as indicated.

Note: Field names must be unique; different record types cannot have common field names.

An example is

```
record complex r:real, i:real end
```

which declares **complex** to be a record type with two fields, r and i, both of which have type **real**.

5.4.2 Creating Records

A record is created during program execution by an expression of the form

```
type value [, value] ...
```

where the type is one declared by **record** and the values are assigned to the fields of the record in the order corresponding to the field names. For example,

```
z := complex 1.0, 2.5
```

assigns to z a **complex** record with a value of 1.0 for the r field and a value of 2.5 for the i field.

Default: An omitted value defaults to &null and is converted to the expected type.

5.4.3 Accessing Records

A record is accessed by field name, using an infix dot notation. Continuing the example above, the value of z.r is 1.0. The infix dot operator binds more tightly than any other infix operator and associates to the left. For example, a.b.c.d and ((a.b).c).d are equivalent.

Records can also be accessed by position like linear arrays. For example, z[1] is equivalent to z.r.

examples:

expressions in sequence	value	signal
z1 := **complex** 0,0	complex	success
z2 := **complex** 3.14, -3.14	complex	success
z1.r	0.0	success
z1.r + z2.i	-3.14	success
z1.r := z2.r	3.14	success
z1.i := z1.r	3.14	success
z1.r - z1.i	0.0	success
z2[2]	-3.14	success
z2[3]		failure

5.5 Copying Structures

Assignment does not copy structures, but rather assigns the same structure to another variable. For example,

```
x := array 10
y := x
```

assigns the same array to x and y. Subsequently, x[3] and y[3] reference the same element of the same array.

A structure may be copied by the built-in function copy(x). For example

```
z := copy(x)
```

assigns a copy of x to z. This copy has the same structure as x and the values of all the elements are the same, but x and z are distinct structures. Subsequently, x[3] and z[3] reference different elements in the corresponding positions of different structures.

5.6 Sorting Structures

The built-in function sort(x,i) produces a copy of x with the elements in sorted order. The array a must be a rectangular, n-by-m array (a linear array is an n-by-1 array). The array is sorted on column i and the elements in the rows corresponding to column i are moved correspondingly.

Default: An omitted value of i defaults to 1.

Error Condition: If i is not in the range 1 <= i <= m, Error 219 occurs.

In sorting, strings are sorted in increasing lexical order (see Section 4.6), while integers and real numbers are sorted in numerical order (see Section 3.1.3 and 3.2.3). The ordering of value of other types is unspecified.

In heterogeneous arrays containing values of different types, values are first sorted by type and then among the values of the same type. The order of types in sorting is

null
integer
real
string
cset
file
procedure
array
table
stack
defined types

Tables are converted to sorted arrays by sort(t,i). The result is an n-by-2 array with the element references in the first column and the corresponding values in the second column. If i is 1, the result is sorted by reference, while if i is 2, the result is sorted by value.

Default: An omitted value of i defaults to 1.

Error Condition: In sort(t,i), if i is not 1 or 2, Error 219 occurs.

5.7 Structure Size

The size of a structure x is the number of elements in it and is the value of size(x).

For arrays, the size is specified by the array prototype and remains constant except for open arrays, in which the size increases as elements are added.

Tables initially have a size of 0, but increase in size as values are assigned to newly referenced elements.

Note: Merely referencing an element in a table does not add that element to the table.

Stacks increase and decrease in size as elements are pushed and popped.

Records are fixed in size by their declaration.

5.8 Generation of Elements in Structures

The operation !x is a generator that produces, as required, successive elements from the structure x. Assignment may be made to !x to change the value of the element.

For linear arrays, the order of generation is from the lower bound to the upper bound. For example, if x is a list

```
every write(!x)
```

writes the elements of x in order from the first to the last.

For arrays of higher dimensionality, the order of generation is by dimension, from left to right. For example, if x is an array with the prototype 3,2,2, the order of generation is

```
x[1,1,1]
x[2,1,1]
x[3,1,1]
x[1,2,1]
x[2,2,1]
x[3,2,1]
x[1,1,2]
x[2,1,2]
x[3,1,2]
x[1,2,2]
x[2,2,2]
x[3,2,2]
```

For tables, the order of generation is unpredictable, but all elements are generated if required.

For stacks, the order of generation is from the top of the stack to the bottom of the stack.

For record types, the order of generation is the same as for linear arrays.

CHAPTER 6

Input and Output

Many aspects of input and output are strongly dependent on specific computer architecture, operating system characteristics, and installation conventions. For these reasons, much of the material in this chapter is machine dependent.

6.1 Files

The term file refers to a set of data that is physically external to the computer itself. Files may be considered to contain a sequence of strings, called lines.

There are two important files that provide for standard program input and output. They permit the program to access data to be processed and provide a mechanism for recording the results of computation. The values of &input and &output are the standard input and output files, respectively.

Error Condition: These keywords are not variables. If an attempt is made to assign a value to one of them, Error 121 occurs.

Note: The method by which standard input and output files are interfaced to a program varies from machine to machine.

6.2 Opening and Closing Files

&input and &output are automatically opened when program execution begins.

A program may, in addition, read data from files other than the standard input file and may write data to files other than the standard output file.

In order to reference files, they are given names. The syntax of file names is machine dependent and varies significantly from one system to another.

A file must be opened to be written or read, and must be closed when input and output are complete. In addition, the status of the file must be established; some files are designated for input and others are designated for output.

All files are automatically closed when program execution is terminated.

Warning: In the case of abnormal program termination, files may not be closed. This can result in the loss of data that has been written to output files. Some systems provide an explicit means of closing files after program termination.

The function open(s1,s2) opens the file with name s1 according to the options specified by s2. The possible options are represented by characters as follows:

r	open for reading
w	open for writing
b	open for reading and writing (bidirectional)
a	open for writing in append mode
l	provide line terminator at end of write
n	do not provide line terminator at end of write
e	provide line terminator after each argument of write
c	character input and output

Notes: Characters in the option sepcification may be duplicated. In the case of mutually exclusive specifications, the last (right-most) specification holds. Not all the options listed above are available on all machines.

In the case of the w option, writing starts at the beginning of the file, causing any data previously contained in the file to be lost. The a option allows data to be written at the end of an existing file. The b option usually applies to interactive input and output at a computer terminal where the terminal behaves like a file that is both written and read.

The l, n, and e options are provided as optional additions to the w, b, and a options. The normal mode of output is l, in which a line terminator is provided at the end of the sequence of strings written by the write function. The n option suppresses this terminator. This permits several strings to be concatenated on the same line with successive calls to the write function. The main use of this option, however, is to provide prompting at a terminal in an interactive mode, allowing the user to respond on the same (visual) line that an inquiry is written. The e mode allows several lines to be written with a single call of the write function.

With the c option, input and output are done one character at a time.

Default: An omitted value of s2 defaults to "r".

Error Condition: If the option specification is invalid, Error 221 occurs.

Failure Condition: open(s1,s2) fails if s1 cannot be opened with the options specified by s2.

The function close(f) closes f. This has the effect of physically completing output (emptying internal buffers used for intermediate storage of data). Once a file has been closed, it must be reopened to be used again. In this case, the file is positioned at the beginning (rewound).

Error Condition: If a file cannot be closed, Error 401 occurs.

6.3 Writing Data to Files

The function write(f,s1,...,sn) writes the strings s1, s2, ..., sn to the file f. The strings are written one after another as a single line, not as separate lines (i.e., they are not separated by line terminators). The effect is as if s1, s2, ..., sn had been concatenated and written as a single line on the file f.

The maximum length of an output line is machine and system dependent, as is the treatment of output lines of excessive length.

Notes: A line terminator is normally added after sn, but see Section 6.4. No actual concatenation is performed by the write function. Since strings output to a file frequently are composed of several parts, the write function may be used to avoid concatenation that otherwise might be necessary. A significant amount of processing time may be saved in this way.

Default: If the first argument to write is not of type **file**, the arguments are written to the standard output file. That is, write(s1,s2,...,sn) writes s1, s2, ..., sn to the standard output file. Note in particular that if the first argument to the write function is a string, it is not automatically converted to type **file**.

Error Condition: If an attempt is made to write on a file that is not open for writing, Error 403 occurs.

Objects of type **integer**, **real**, **cset**, **file**, and **null** are automatically converted to string as described in Section 4.4. Arguments of other types are converted to **string** by use of the image function (see Section 4.4). Thus arguments of any type can be specified in the write function.

Notes: If the first argument to write is a file, it is not converted to string, but rather used as the output file. For arguments of type file is positions other than the first, the image function converts a file to a string with the corresponding file name. Single enclosing quotes are used to distinguish the images of files.

examples:

expressions in sequence	output	file
out := open("data.txt")		
flag := "****"		
sep := ":"		
write(out)	ø	data.txt
write(out,flag,"a",sep,"b")	****a:b	data.txt
write(flag,"a",sep,"b")	****a:b	&output
write(out,"x",sep,"y",sep,"z",flag)	x:y:x****	data.txt
write(1,sep,2.0,sep,"2")	1:2.0:2	&output
write(sep,out,sep)	:'data.txt':	&output
write(out,out)	'data.txt'	data.txt

6.4 Reading Data from Files

The function read(f) reads the next line of data from the file f.

Failure Condition: When the end of a file is reached (that is, when there are no more lines in the file), read(f) fails.

Default: An omitted value for f defaults to &input.

read(f) is a generator, producing successive line of input as required. For example,

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

copies all the lines in the standard input file to the standard output file.

Error Conditions: If an input line exceeds 100 characters, Error 411 occurs. If an attempt is made to read from a file which is not opened for reading, Error 403 occurs. See also Section 9.1.1.

CHAPTER 7

Miscellaneous Operations

7.1 Random Number Generation

The value of random(i) is an integer taken from a pseudo-random sequence with the range 1 <= j <= i.

The pseudo-random sequence is generated by a linear congruence relation starting with an initial seed value of 0. This sequence is the same from one program execution to another, allowing program testing in a reproducible environment. The seed may be changed by an assignment to &random. For example,

```
&random := 0
```

resets the seed to its initial value.

Note: The maximum range of values in the pseudo-random sequence and the maximum seed value are machine dependent.

Error Condition: If the value of i in random(i) or the value assigned to &random is negative or out of range, Error 207 occurs.

7.2 Time and Date

The value of the keyword &date is the current date in the form mm/dd/yy. For example, the value of &date for April 1, 1979 is 04/01/79.

The value of the keyword &clock is the current time of day in the form hh:mm:ss. For example, the value of &clock for 8:00 p.m. is 20:00:00.

The value of the keyword &time is the elapsed time in milliseconds starting at the beginning of program execution.

Note: The time required for program compilation is not included in the value of &time.

Error Condition: &date, &clock, and &time are not variables. If an attempt is made to assign a value to one of them, Error 121 occurs.

CHAPTER 8

Procedures

The basic unit of a program is the procedure. All computation is performed by invoking procedures.

8.1 Procedure Declaration

A procedure is declared in the form

```
procedure name [( argument-list ) ] [: return-type]
   [identifier-declaration] [, identifier-declaration] ...
   [default-declaration]
   [initial-section]
   [procedure-body]
end
```

The argument list specifies the arguments of the procedure and has the form

```
argument [, argument] ...
```

Each argument consists of an identifier and an optional type specification:

```
identifier [: type]
```

Default: An omitted type defaults to **any**.

The return type specifies the type of the value returned by the procedure.

Default: An omitted return type defaults to **any**.

An identifier declaration has the form

```
[scope] [retention] identifier [, identifier] ... [: type]
```

The scope declaration may be either **local** or **global**. The value of a local identifier is accessible only to a specific invocation of a procedure. The value of a global identifier is available to all invocations of all procedures in which the global identifier appears. Identifiers in the argument list are local.

The retention specification is allowed only for local identifiers and may be either **dynamic** or **static**.

Defaults: An omitted retention specification defaults to **dynamic**. An omitted type specification defaults to **any**.

Dynamic identifiers exist only during each invocation of the procedure. Static identifiers come into existence at the first call of the procedure in which they are declared and remain in existence after return from the procedure so that their values are retained between calls of the procedure.

The default declaration has the form

default default-type

where the default type may be **local**, **global**, or **error**. The default declaration determines the handling of identifiers in the procedure that are not declared and are not among the global identifiers of other procedures. If the default is **local**, such identifiers are treated as if they had been declared local. Similarly, if the default is **global**, such identifiers are treated as if they had been declared global. If the default is **error**, such identifiers are treated as errors.

The initial section has the form

initial expr

The expression in the initial section is evaluated once when the procedure is first called. The initial section is useful for assigning values to global or static local identifiers.

The procedure body consists of a sequence of expressions that are executed when the procedure is called.

Some examples of procedure declarations follow.

```
procedure max(i:integer,j:integer):integer
   if i > j then return i else return j
end

procedure accum(s:string):string
   local static t:string
   initial t := ","
   t := t || s || ","
   return t
end
```

8.2 Procedure Activation

8.2.1 Procedure Invocation

Procedures are invoked in the same form that built-in functions are called:

name (expr [, expr] ...)

For example, the procedure max given in the example above might be used as follows:

```
m := max(size(x),size(y))
```

Argument transmission is by value. When a procedure is called, the expressions given in the call are evaluated from the left to the right.

The values of the expressions in the call are assigned to the corresponding identifiers in the argument list of the procedure. Control is then transferred to the first expression in the procedure body.

Failure Condition: If any expression in the call fails, the remaining expressions are not evaluated, the procedure is not called and the calling expression fails.

Note: If more expressions are given in the call than are specified in the procedure declaration, the excess expressions are evaluated, but their values are discarded. If fewer expressions are given in the call than are specified in the procedure declaration, &null is provided for the remaining arguments.

8.2.2 Return from Procedures

When a procedure is called, the expressions in the procedure body are executed until a return expression is encountered. There are four forms of return expression:

return [expr]
succeed [expr]
fail
suspend [expr]

Defaults: An omitted expr in a return expression defaults to &null. An implicit **return** &null is provided at the end of every procedure body.

1. The expression **return** expr terminates the call of a procedure and returns the result of evaluating expr. If expr fails, the procedure call fails. Otherwise the value of expr becomes the value of the calling expression. For example

```
j := max(size(x),size(y))
```

assigns to j the size of the larger of the two objects x and y.

2. The expression **succeed** expr is the same as **return** expr, except that the signal resulting from the evaluation of expr is ignored and the procedure signals success.

Note: If expr fails, &null is returned.

3. The expression **fail** terminates the call of a procedure with a failure signal, causing the calling expression to fail. Consider the following procedure.

```
procedure typeq(x,y)
   if type(x) == type(y) then succeed else fail
end
```

This procedure compares the types of x and y, succeeding if they are the same and failing otherwise.

4. The expression **suspend** expr is similar to **succeed** expr, except that the procedure call is left in suspension so that it may be resumed for additional computation. Execution of the procedure body is resumed if goal-directed evaluation requests another alternative. Thus suspended procedures are generators. Consider the following procedure.

```
procedure timer(t):integer
   while &time < t do suspend
   fail
end
```

This procedure suspends evaluation until the time exceeds a specified limit, in which case it fails. Therefore

```
every timer(&time + 1000) do expr
```

evaluates expr repeatedly during an interval of approximately 1000 milliseconds.

suspend, like **every**, produces all alternatives of expr as required. For example

```
suspend (1|2|3)
```

suspends with the values 1, 2, and 3 on successive activations of the procedure in which it appears. If the procedure is activated again, evaluation continues with the expression following the **suspend**.

If the expression in **return** or **succeed** is a global identifier or a computed variable (such as an array reference), the variable is returned. In the case of local identifiers, only the value is returned. An assignment can be made to the result of a procedure call that returns a variable. Consider the following procedure:

```
procedure maxel(x:array,i:integer,j:integer)
   if x[i] > x[j] then return x[i]
      else return x[j]
end
```

An assignment can be made to a call of this procedure to change the value of the maximum of the elements i and j in x:

```
maxel(roster,k,m) := n
```

Unlike **return** and **succeed**, **suspend** returns a local identifier as a variable, since local identifiers in a procedure remain in existence while the procedure is suspended.

8.2.3 Procedure Level

Since procedures can invoke other procedures before they return, several procedures may be invoked at any one time. The value of the &level is the number of procedures that have currently been invoked.

Error Condition: There is no specific limit to the number of procedures that may be invoked at any one time, but storage is required for procedure invocations that have not returned. If available storage is exhausted, Error 501 occurs.

8.2.4 Tracing Procedure Activity

Tracing of procedure invocation is controlled by the keyword &trace. If the value of &trace is not zero, a diagnostic message is printed on the standard output file each time a procedure is called and each time a procedure returns or suspends. The value of &trace is decremented for each trace message.

Default: The initial, default value of &trace is 0.

Note: Tracing stops automatically when &trace is decremented to 0. If a negative value is assigned to &trace, tracing continues indefinitely.

In the case of a procedure call, the trace message includes the name of the procedure and the values of its arguments. The message is indented with a number of dots equal to the level from which the call is made (&level). In the case of procedure return, the trace message includes the function name, the type of return, and the value returned, except in the case of failure. The indentation corresponds to the level to which the return is made.

An example is given by the following program:

```
procedure acker(m,n)
  if (m|n) < 0 then fail
  if m = 0 then return n + 1
  if n = 0 then return acker(m-1,1)
  return acker(m-1,acker(m,n-1))
end

procedure main
  &trace := -1
  acker(1,3)
end
```

The trace output produced by this program is

```
.line 10: acker(1,3)
..line 5: acker(1,2)
...line 5: acker(1,1)
....line 5: acker(1,0)
.....line 4: acker(0,1)
.....line 3: acker returned 2
....line 3: acker returned 2
....line 3: acker(0,2)
....line 3: acker returned 3
...line 3: acker returned 3
...line 3: acker(0,3)
...line 3: acker returned 4
..line 3: acker returned 4
..line 3: acker(0,4)
..line 3: acker returned 5
.line 3: acker returned 5
line 3: main returned &null
```

8.3 Listing Identifier Values

The function display(i) prints a list of all identifiers and their values in the i levels of procedure invocation starting at the current procedure invocation.

Default: An omitted value of i defaults to 1 (only the identifiers in the currently invoked procedure are displayed).

Note: display(&level) displays the identifiers in all procedure invocations leading to the current invocation.

As an example of the display of identifiers, consider the following program:

```
procedure hex(x) global hexd
  display(&level)
  return &ascii[16 * find(x[1],hexd) +
    find(x[2],hexd) - 16]
end

procedure main local label; global hexd
  hexd := "0123456789ABCDEF"
  label := "hex(61)="
  write(label,hex("61"))
end
```

The output of display(&level) is

```
hex locals:
   x = "61"
main locals:
   label = "hex(61)="
program globals:
   main = procedure main
```

```
hex = procedure hex
hexd = "0123456789ABCDEF"
```

The program globals, which are common to all procedures, are listed at the end of every display output, regardless of whether or not the globals are referenced by the displayed procedures.

8.4 Procedure Names and Values

A procedure declaration establishes an object of type **procedure** as the initial value of the identifier that is the procedure name. This object can be assigned to another variable and the procedure can be called using the new variable. For example

```
imax := max
```

assigns to imax the procedure for max as given earlier. Subsequently,

```
imax(i,j)
```

can be used to compute the maximum of i and j.

Any expression that produces a value of type **procedure** may be used in a call. For example, if procs is a list whose elements have procedures as value, such as

```
procs[1] := max
```

then

```
procs[1](i,j)
```

computes the maximum of i and j.

If the name of a declared procedure is the same as the name of a built-in function, the declaration overrides the built-in meaning.

Identifiers that are the names of built-in functions have objects of type **procedure** as their value. These values may be assigned to other variables and used in the same fashion as declared procedures.

Identifiers that are the names of built-in functions and declared procedures are not variables and values cannot be assigned to them.

Note: Such names are similar to those keywords, such as &time, that are not variables.

Error Condition: If an attempt is made to assign a value to an identifier that is the name of a built-in function or declared procedure, Error 121 occurs.

CHAPTER 9

Programs

9.1 Program Structure

A program is a sequence of declarations for records and procedures. These declarations may appear in any order. Every program must contain a procedure named main.

9.1.1 Preparation of Program Text

A program is essentially a file. Program files may be constructed using any of the utilities available, which vary from installation to installation. Installations with interactive facilities may allow the program to be entered and run directly from a terminal, although this is impractical for all but the shortest programs.

As a file, a program is a sequence of lines. In most cases it is convenient and natural to parallel the logical structure of a sequence of expressions by the physical structure of a sequence of lines.

If desired, several expressions can be placed on a single line using semicolons to separate them. For example

```
x := 1
y := 2
z := 0
```

can also be written as

```
x := i; y := 2; z := 0
```

The maximum length of a program line is 120 characters, although some systems may impose more stringent limits. Sometimes an expression is too long to fit on a line. An expression may be split between lines at any point that a blank may be used. Infix operators whose operands span lines must be surrounded by blanks.

Warning: Care should be taken not to split expressions at places where components are optional. For example

```
return e
```

and

```
return
   e
```

are quite different.

A string literal may be continued from one line to the next by entering an underscore (_) as the last character of the current line. When a line is continued in this way, the underscore as well as any blanks or tab characters at the beginning of the next line are ignored to allow normal indentation and visual layout conventions to be used.

9.1.2 Program Character Set

Different computers, systems, and peripheral equipment use different character sets and vary in the characters that are available on input and output. As mentioned in Sections 1.3 and 4.1, Icon uses the ASCII character set internally. For systems that do not use this character set, Icon provides automatic translation between the external character set of the system on which it is run and the internal character set that it uses. Since some systems are limited in the number of characters available and methods for their graphic representation, certain characters are equivalent for syntactic purposes. These are:

```
lower-case letters and upper-case letters
tab and blank
~ and ^
` and @
[ and {
] and }
| and \ and !
```

Note: In literal strings, all characters are distinct and the equivalences above do not apply.

Although this manual distinguishes between these characters to make program examples more readable, the alternates above can be used without affecting program behavior. For example, the following expressions are equivalent:

```
if i := upto(c) then {write(i); x[i] := c}

IF I := UPTO(C) THEN [WRITE(I); X[I] := C]

IF i := Upto(c) Then {wrIte(I); x{i] := C}
```

While the last example is equivalent to the others, it is clearly good practice to be consistent in the use of alternative characters.

9.1.3 Comments

A comment is text in the line of a program that is not part of the program itself, but is included to describe the program or to provide other auxiliary information. The character # causes the rest of the line on which it appears to be treated as a comment.

The following program segment shows the use of comments.

```
#       ======                                  =====
#       ====== C H A R A C T E R    S E T S =====
#       ======                                  =====

#       These functions perform the operation of union, intersection,
#       and difference on two character sets.

procedure union(cs1,cs2)          # union of cs1 and cs2
  return cset(cs1 || cs2)
end

procedure inter(cs1,cs2)          # intersection of cs1 and cs2
  return ~(~cs1 || ~cs2)
end

procedure differ(cs1,cs2)         # difference of cs1 and cs2
  return ~(~cs1 || cs2)
end
```

9.2 Including Text from Other Files

Text from other program files can be included by the declaration

include file-name

The contents of the named file replace the **include** declaration. This provides a convenient mechanism for incorporating procedures or record declarations from libraries.

Notes: **include** may not appear inside a procedure or record declaration. Included files may contain other **include** declarations. It is good practice, although not necessary, to have included files consist of complete declarations.

9.3 Program Execution

There are two phases of program execution. During the first phase, the text of the program is translated into a form that can be executed by the computer. The program is then executed to carry out the operations that it specifies.

9.3.1 Translation Errors

The translator can detect a variety of errors. Most of the errors that the translator can detect are syntactic ones: illegal grammatical constructions. The translator can also detect a few semantic errors, such as undeclared identifiers in a procedure in which **default error** is specified.

The translator lists errors and the location at which they are detected. See Appendix E for a list of error messages.

Notes: Some grammatical errors are not detected until after the location of the actual cause of the error. For example, if an extra left brace appears in an expression, the error is not detected until some construction occurs that requires the matching, but missing right brace. As a result of this phenomenon, the translator message may not properly indicate the cause or location of the error. Similarly, some kinds of errors may cause the translator to mistakenly interpret subsequent constructions as erroneous when, in fact, they are correct. Several diagnostic messages referring to locations in proximity should be suspect.

If the translator detects an error, the translation process is continued, but the program is not executed.

9.3.2 Initiating Execution

Program execution begins by invocation of the procedure named main.

Error Condition: If the program contains no procedure with the name main, Error 108 occurs.

9.3.3 Program Termination

Program execution terminates automatically on return from the initial call of the main procedure.

Note: Since a default **return** is provided at the end of every procedure body, program execution terminates on completion of evaluation of the body of the main procedure, even if no explicit return has been made.

Program termination is also caused by evaluation of stop(f,s1,s2,...,sn). The function stop writes s1, s2, ..., sn to f in the fashion of the write function (see Section 6.2) and then causes termination.

Note: The stop function can be used to terminate program execution at an arbitrary place and is a convenient way of handling errors or abnormal conditions that are detected by a program.

The function exit() terminates program execution and preserves the core image of the program so that it can be saved and restarted at a future time.

Note: The capability for saving and restarting core images, as well as the method by which it is done, is machine dependent. The exit function may not be available on some machines.

There are three kinds of errors that may occur during program execution: program errors, processor errors, and exception errors.

Program errors result from logical mistakes, invalid data, and so forth. If one of these errors occurs, an error number and an explanatory message are printed and program execution is terminated. Program errors are listed in Appendix E.

Processor errors occur in the case of an unexpected situation or internal inconsistency in the Icon processor. Such errors cause program termination with a message and a description of the internal problem. If a processor error occurs, the problem should be brought to the attention of the person responsible for the maintenance of the Icon processor. It is advisable to leave the program that caused the problem, as well as any data that was processed, intact so that tests can be performed to locate the cause of the error.

Exception errors may occur for a variety of reasons that are machine dependent. For example, arithmetic overflow on some computers causes termination without allowing the Icon processor the opportunity to gain control. When a program terminates abnormally due to an exception error, the usual termination messages are not provided and files may not be closed. See Section 6.4.

APPENDIX A

Syntax

A.1 Formal Syntax

In the following listing of the formal syntax of Icon, the syntactic types bar, period, left-bracket, and right-bracket indicate occurrences of the characters |, ., [, and], which have metalinguistic uses in the syntax description language. The lexical types identifier, integer-literal, real-literal, string-literal, file-name, and word are not described here. See their description in the body of the manual. The syntactic type record-type is determined by record declarations and varies from program to program.

```
        program ::= [declar [declar] ... ]

         declar ::= include | record | procedure

        include ::= include file-name

         record ::= record field-list end

     field-list ::= [identifier [: type] [, identifier [: type] ... ]

           type ::= integer | real | string | cset | file | procedure
                  | array | table | stack | null | any | record-type

      procedure ::= proc-header proc-declar initial-section
                    proc-body end

    proc-header ::= procedure identifier [( argument-list )]
                    [: return-type]

  argument-list ::= [argument [, argument]]

       argument ::= identifier [: type]

    return-type ::= type

     proc-declar ::= [default-declar] [ident-declar [, ident-declar] ... ]

 default-declar ::= default default-type

   default-type ::= local | global | error

   ident-declar ::= global ident-list [: type] | local [retention]
                    ident-list [: type]

     ident-list ::= identifier [, identifier] ...
```

```
      retention ::= static | dynamic

initial-section ::= [initial expr]

      proc-body ::= [expr [; expr] ... ]

           expr ::= literal | keyword | operation | call | reference
                  | list | structure | control-struct | compound-expr
                  | (expr)

        literal ::= integer-literal | real-literal | string-literal

        keyword ::= & word

      operation ::= prefix-oper expr | expr suffix-oper | expr
                    infix-oper expr

    prefix-oper ::= + | - | ~ | !

    suffix-oper ::= + | -

     infix-oper ::= ? | & | := | :=: | bar | = | ~= | > | < | >=
                  | <= | == | ~== | bar bar | + | - | * | / | ** |

           call ::= expr ( expr-list )

      expr-list ::= expr [, expr] ...

      reference ::= expr left-bracket expr-list right-bracket
                  | expr period identifier

           list ::= < expr-list >

      structure ::= array | table | stack | record-object

          array ::= array [array-prototype] [type] [initial-clause]

array-prototype ::= range [, range] ...

          range ::= [lower-bound :] upper-bound

    lower-bound ::= expr

    upper-bound ::= expr

 initial-clause ::= initial expr

          table ::= table [size] [ref-type] [, value-type]

           size ::= expr

       ref-type ::= type

     value-type ::= type

          stack ::= stack [size] [type]
```

```
record-object ::= record-type expr-list

control-struct ::= if-then-else | while-do | until-do | every-do | repeat
                 | fails | null | to-by | next | break

if-then-else ::= if expr then expr [else expr]

while-do ::= while expr do expr

until-do ::= until expr do expr

every-do ::= every expr [do expr]

repeat ::= repeat expr

fails ::= expr fails

null ::= null expr

to-by ::= expr to expr [by expr]

next ::= next

break ::= break

compound-expr ::= { [expr [; expr] ... ] }
```

A.2 Precedence and Associativity

The relative precedence of reserved words and operators, arranged in ascending order, follows. For infix operators, the associativity is listed also.

	precedence	type	associativity
null	1		
if-then-else	1		
while-do	1		
until-do	1		
every-do	1		
repeat	1		
fail	1		
succeed	1		
return	1		
suspend	1		
?	2	infix	left
&	3	infix	left
fails	4		
:=	5	infix	right
:=:	5	infix	right
to-by	6		
|	7	infix	left
=	8	infix	left
~=	8	infix	left

<	8	infix	left
<=	8	infix	left
>	8	infix	left
>=	8	infix	left
==	8	infix	left
~==	8	infix	left
||	9	infix	left
+	10	infix	left
-	10	infix	left
*	11	infix	left
/	11	infix	left
**	12	infix	right
+	13	suffix	
-	13	suffix	
~	14	prefix	
+	14	prefix	
-	14	prefix	
=	14	prefix	
.	15	infix	left

A.3 Reserved Words

The following reserved words cannot be used as identifiers:

array	**dynamic**	**fails**	**integer**	**record**	**succeed**	**while**
break	**else**	**file**	**local**	**repeat**	**suspend**	
by	**end**	**global**	**next**	**return**	**table**	
cset	**error**	**if**	**null**	**stack**	**then**	
default	**every**	**include**	**procedure**	**static**	**to**	
do	**fail**	**initial**	**real**	**string**	**until**	

A.4 The Significance of Blanks

As a general rule, blanks are syntactic separators (except that a blank in a string literal represents the blank character and has no syntactic significance). Syntactically, blanks are mandatory in some places and optional in others.

Blanks are mandatory where they are necessary to avoid ambiguities:

(1) Between reserved words and expressions unless the expressions are enclosed in parentheses.

(2) Surrounding infix operators that otherwise would be adjacent to prefix or suffix operators. If a blank occurs on one side of an infix operator, it must occur on the other size as well.

Blanks are optional before and after punctuation characters such as parentheses, braces, and commas.

APPENDIX B

Built-In Operations

The following sections list the built-in operations of Icon. The primary section references are cited.

B.1 Functions

function	section
any(c,s,i,j)	4.7.2
bal(c1,c2,c3,s,i,j)	4.7.2
center(s1,i,s2)	4.5.3
close(x)	5.1.3, 5.2.3, 6.4
copy(x)	5.5
cset(x)	4.3
display(i)	8.3
exit()	9.3.3
find(s1,s2,i,j)	4.7.1
image(x)	4.4
integer(x)	3.4.1
left(s1,i,s2)	4.5.3
lge(s1,s2)	4.6
lgt(s1,s2)	4.6
lle(s1,s2)	4.6
llt(s1,s2)	4.6
many(c,s,i,j)	4.7.2
map(s1,s2,s3)	4.5.5
match(s1,s2,i,j)	4.7.1
mod(i,j)	3.1.2
move(i)	4.8.2
numeric(x)	3.5
open(x,s)	5.1.3, 6.4
pop(k)	5.3.2
pos(i,s)	4.2.5
push(k,x)	5.3.2
random(i)	7.1
real(x)	3.4.2
read(f)	6.4
repl(s,i)	4.5.2
reverse(s)	4.5.5
right(s1,i,s2)	4.5.3
section(s,i,j)	4.5.4
size(x)	4.2.3, 5.7
sort(x,i)	5.6
stop(f,s1,s2,...,sn)	9.3.3
string(x)	4.4
substr(s,i,j)	4.5.4
tab(i)	4.8.2
top(k)	5.3.2
trim(s,c)	4.5.5

type(x)	4.4
upto(c,s,i,j)	4.7.2
write(f,s1,s2,...,sn)	6.2

B.2 Operators

B.2.1 Infix Operators

operator	section
?	4.8.4
:=	2.3
:=:	2.3
|	2.8.3
&	2.8.4
+	3.1.2
-	3.1.2
*	3.1.2
/	3.1.2
**	3.1.2
=	3.1.3
~=	3.1.3
>	3.1.3
>=	3.1.3
<	3.1.3
<=	3.1.3
||	4.5.1
==	4.6
~==	4.6
.	5.4.3

B.2.2 Prefix Operators

operator	section
+	3.1.2
-	3.1.2
~	4.3
!	4.5.4, 5.8
=	4.8.3

B.2.3 Suffix Operators

operator	section
+	3.1.2
-	3.1.2

B.3 Keywords

keyword	section
&ascii	4.2.2
&clock	7.2
&date	7.2
&input	6.1
&lcase	4.2.2
&level	8.2.3
&null	2.1
&output	6.1
&pos	4.8.1
&random	7.1
&subject	4.8.1
&time	7.2
&trace	8.2.4
&ucase	4.2.2

APPENDIX C

Summary of Defaults

C.1 Initial Values of Identifiers

Omitted type specifications default to **any**. When a type is specified for an identifier the initial, default value of the identifier is the value that results from converting an object of type **null** to the type specified for the identifier. The result is the same as for implicit or explicit conversion of type **null** to a specified type, and produces the following results:

type	initial value
integer	0
real	0.0
string	""
cset	cset("")
file	standard output file
procedure	error termination procedure
array	open(**array** 0)
table	**table** 0
stack	**stack** 0
null	&null
any	&null

The default initial value for a record is an object with default initial values for each field according to its type.

C.2 Omitted Arguments in Functions

any(c)	any(c,&subject,&pos,0)
bal(,,,s,i,j)*	bal(cset(&ascii),cset("("),cset(")"),s,i,j)
bal(c1,c2,c3)*	bal(c1,c2,c3,&subject,&pos,0)
center(s,i)	center(s,i," ")
display(i)	display(1)
find(s1,s2)	find(s1,s2,1,0)
find(s)	find(s,&subject,&pos,0)
left(s,i)	left(s,i," ")
many(c,s)	many(c,s,1,0)
many(c)	many(c,&subject,&pos)
match(s1,s2)	match(s1,s2,1,0)
match(s)	match(s,&subject,&pos,0)
open(s)	open(s,"r")
pos(i)	pos(i,&subject)
read()	read(&input)
right(s,i)	right(s,i," ")

```
section(s)      section(s,1,0)
sort(x)         sort(x,1)
trim(s)         trim(s,cset(" "))
upto(c,s)       upto(c,s,1,0)
upto(c)         upto(c,&subject,&pos,0)
```

*These defaults apply separately and may be used in any combination. For example, bal() defaults to

```
bal(cset(&ascii),cset("("),cset(")"),&subject,&pos,0)
```

In addition, if c1 is omitted, the balanced string may end at any position, including at the end of s.

Omitted arguments otherwise default to &null and are converted to the expected types accordingly. For example, find(s1,s2,2) defaults to find(s1,s2,2,0).

C.3 Omitted Components in Structure Specifications

When optional components are omitted in structure specifications, the following defaults are used:

```
array           array 0 any initial &null
table           table 0 any, any
stack           stack 0 any
```

APPENDIX D

Summary of Type Conversions

D.1 Explicit Conversions

The following explicit type conversions are supported:

cset(x)	**cset, integer, null, real,** and **string**
integer(x)	**cset, integer, null, real,** and **string**
real(x)	**cset, integer, null, real,** and **string**
string(x)	**cset, file, integer, null, real,** and **string**

The success of a conversion operation usually depends on the specific value involved. For example, integer("10") succeeds, but integer("1a") fails.

D.2 Implicit Conversions

Where required by context, implicit conversions are performed automatically for all types corresponding to the type-conversion functions listed above. In addition, all types are automatically converted to **null** by the **null** control structure.

APPENDIX E

Summary of Error Messages

E.1 Translator Error Messages

A list of translator error messages follows. These messages indicate the erroneous condition detected by the translator, not necessarily the cause of the error.

assignment to nonvariable
cannot open include file
duplicate declaration for local identifier
extraneous closing brace
extraneous end
identifier too long
integer character larger than base
invalid character
invalid construction
invalid context for break
invalid context for next
invalid default
invalid escape specification
invalid expression list
invalid field name
invalid integer base
invalid integer literal
invalid keyword
invalid keyword construction
invalid operator
invalid real literal
missing argument
missing closing parenthesis
missing declaration
missing do in while or until expression
missing expression list
missing procedure end
missing procedure name
missing quote
missing record end
missing record field
missing record name
missing semicolon
missing then in if-then expression
multiply declared field name
numeric literal too long
string literal too long
unclosed list
undeclared identifier
unexpected end-of-file

E.2 Program Error Messages

Program errors fall into several major classifications, depending on the nature of the error. Error numbers are composed from the number of the category times 100 plus a specific identifying number within the category. In the list that follows, omitted numbers are reserved for possible future use.

Category 1: invalid type or form

101 integer expected
102 real expected
103 numeric expected
104 string expected
105 cset expected
106 file expected
107 procedure expected
108 record expected
109 stack expected
110 array expected
111 invalid type to size
112 invalid type for reference
113 invalid type to close
114 invalid type to sort
121 variable expected

Category 2: invalid argument or computation

201 division by zero
202 zero second argument to mod
203 integer overflow
204 real overflow
205 real underflow
206 negative first argument in real exponentiation
207 invalid value to random or &random
211 negative second argument to repl
212 negative second argument to left
213 negative second argument to right
214 negative second argument to center
215 second and third arguments to map of unequal length
216 erroneous array bounds
217 negative stack size
218 negative table size
219 invalid second argument to sort
221 invalid option for open

Category 3: invalid structure operation

301 table size exceeded
302 stack size exceeded

Category 4: input/output errors

401 cannot close file
402 attempt to read file not open for reading
403 attempt to write file not open for writing
411 input string too long

Category 5: capacity exceeded

501 insufficient storage

APPENDIX F

The ASCII Character Set

F.1 Characters and Codes

position	code	graphic	keyboard entry	control function
1	000		control shift P	null
2	001		control A	
3	002		control B	
4	003		control C	
5	004		control D	
6	005		control E	
7	006		control F	
8	007		control G	bell
9	010		control H	backspace
10	011		control I	horizontal tab
11	012		control J	line feed
12	013		control K	vertical tab
13	014		control L	form feed
14	015		control M	carriage return
15	016		control N	
16	017		control O	
17	020		control P	
18	021		control Q	
19	022		control R	
20	023		control S	
21	024		control T	
22	025		control U	
23	026		control V	
24	027		control W	
25	030		control X	
26	031		control Y	
27	032		control Z	
28	033		control shift K	escape
29	034		control shift L	
30	035		control shift M	
31	036		control shift N	
32	037		control shift O	
33	040		space	
34	041	!	!	
35	042	"	"	
36	043	#	#	
37	044	$	$	
38	045	%	%	
39	046	&	&	
40	047	'	'	

position	code	graphic	keyboard entry	control function
41	050	(	(	
42	051	)	)	
43	052	*	*	
44	053	+	+	
45	054	,	,	
46	055	-	-	
47	056	.	.	
48	057	/	/	
49	060	0	0	
50	061	1	1	
51	062	2	2	
52	063	3	3	
53	064	4	4	
54	065	5	5	
55	066	6	6	
56	067	7	7	
57	070	8	8	
58	071	9	9	
59	072	:	:	
60	073	;	;	
61	074	<	<	
62	075	=	=	
63	076	>	>	
64	077	?	?	
65	100	@	@	
66	101	A	shift A	
67	102	B	shift B	
68	103	C	shift C	
69	104	D	shift D	
70	105	E	shift E	
71	106	F	shift F	
72	107	G	shift G	
73	110	H	shift H	
74	111	I	shift I	
75	112	J	shift J	
76	113	K	shift K	
77	114	L	shift L	
78	115	M	shift M	
79	116	N	shift N	
80	117	O	shift O	
81	120	P	shift P	
82	121	Q	shift Q	
83	122	R	shift R	
84	123	S	shift S	
85	124	T	shift T	
86	125	U	shift U	
87	126	V	shift V	
88	127	W	shift W	
89	130	X	shift X	
90	131	Y	shift Y	
91	132	Z	shift Z	

position	code	graphic	keyboard entry	control function
92	133	[	[	
93	134	\	\	
94	135	]	]	
95	136	^	^	
96	137			
97	140	`	`	
98	141	a	A	
99	142	b	B	
100	143	c	C	
101	144	d	D	
102	145	e	E	
103	146	f	F	
104	147	g	G	
105	150	h	H	
106	151	i	I	
107	152	j	J	
108	153	k	K	
109	154	l	L	
110	155	m	M	
111	156	n	N	
112	157	o	O	
113	160	p	P	
114	161	q	Q	
115	162	r	R	
116	163	s	S	
117	164	t	T	
118	165	u	U	
119	166	v	V	
120	167	w	W	
121	170	x	X	
122	171	y	Y	
123	172	z	Z	
124	173	{	{	
125	174	|	|	
126	175	}	}	
127	176	~	~	
128	177		rub out	delete

Acknowledgement

The Icon programming language was designed by Dave Hanson and Tim Korb in collaboration with the author. In addition, Walt Hansen has made a number of valuable contributions to the design. The author is indebted to Dave Hanson, Tim Korb, Walt Hansen, and Madge Griswold for careful readings of drafts of this manual and for advice on the presentation of language features.

References

1. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey. 1971.

2. Griswold, Ralph E., David R. Hanson, and John T. Korb. An Overview of the SL5 Programming Language. SL5 Project Document S5LD1d, The University of Arizona, Tucson, Arizona. October 18, 1977.

3. American National Standards Institute. USA Standard Code for Information Interchange, X3.4-1968. New York, New York. 1968.

4. Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language. Prentice-Hall, Inc. Englewood Cliffs, New Jersey. 1978.

5. IBM Corporation. System/370 Reference Summary. Form GX20-1850-3. White Plains, New York. 1976.

6. Control Data Corporation. SCOPE Reference Manual. Publication Number 60307200. Sunnyvale, California. 1971.

```
      retention ::= static | dynamic

initial-section ::= [initial expr]

      proc-body ::= [expr [; expr] ... ]

           expr ::= literal | keyword | operation | call | reference
                  | list | structure | control-struct | compound-expr
                  | (expr)

        literal ::= integer-literal | real-literal | string-literal

        keyword ::= & word

      operation ::= prefix-oper expr | expr suffix-oper | expr
                    infix-oper expr

    prefix-oper ::= + | - | ~ | !

    suffix-oper ::= + | -

     infix-oper ::= ? | & | := | :=: | bar | = | ~= | > | < | >=
                  | <= | == | ~== | bar bar | + | - | * | / | ** |

           call ::= expr ( expr-list )

      expr-list ::= expr [, expr] ...

      reference ::= expr left-bracket expr-list right-bracket
                  | expr period identifier

           list ::= < expr-list >

      structure ::= array | table | stack | record-object

          array ::= array [array-prototype] [type] [initial-clause]

array-prototype ::= range [, range] ...

          range ::= [lower-bound :] upper-bound

    lower-bound ::= expr

    upper-bound ::= expr

 initial-clause ::= initial expr

          table ::= table [size] [ref-type] [, value-type]

           size ::= expr

       ref-type ::= type

     value-type ::= type

          stack ::= stack [size] [type]
```