Reference Manual for the Icon Programming Language Version 5 (C Implementation for UNIX)*

Gary A. Coutant, Ralph E. Griswold, and Stephen B. Wampler

IR 81-4a

December 1981, Corrected July 1982

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS79-03890

CONTENTS

Chapter 5 — Strings and Character Sets

Chapter 6 — Basic String Operations

Chapter 7 String Scanning

Chapter 8 Structures

Chapter 9 Input and Output

Chapter 10 Miscellaneous Operations

Chapter 11 Procedures

Chapter 12 Program Preparation

Chapter 13 Programming Considerations

Chapter 1

Introduction

1.1 Background

Icon is the most recent in a series of programming languages that started with SNOBOL [1] SNOBOL was a very simple language with only one data type, the string, and a few pattern-matching statements expressed in a rigid syntax The syntax of SNOBOL was primitive and its only control structure was the goto, which could be conditional on the success of pattern matching and in which the target label could be computed One exotic feature of SNOBOL was its ability to construct identifiers during program execution and reference their values indirectly

SNOBOL 2 [2] which was in use for only a short period of time, was a minor refinement of SNOBOL SNOBOL 3 [3] extended the original SNOBOL language with a repertoire of built-in functions and a mechanism for programmer-defined procedures The concept of success and failure was generalized to include a variety of comparison and testing operations SNOBOL 3 retained the single string data type, static pattern matching and primitive control structures of the original SNOBOL SNOBOL 3 is still in limited use

SNOBOL 4 [4] departed more radically from the earlier languages in the series It introduced a variety of data types and the ability to construct and manipulate patterns as data objects dynamically during program execution Along with this facility the pattern matching repertoire was substantially increased Arrays tables and defined types (records) in SNOBOL4 added the ability to produce and process structures Tables at the same time provided a facility for associative reference of a more disciplined type than the indirect referencing facility of SNOBOL although the latter was retained in SNOBOL4 An esoteric feature, originally planned for SNOBOL was realized for the first time in SNOBOL 4 run-time compilation allowed strings to be converted into executable code in the course of program execution Despite the advances in facility SNOBOL 4 retained the primitive control structures of the earlier languages Because of new data types and operations SNOBOL 4 is best characterized as a general-purpose language with a strong emphasis on string processing whereas the earlier languages were special-purpose string processing languages SNOBOL 4 is in wide use at the present time for a variety of applications [5]

SL 5 ("SNOBOL Language 5") [6] was an even more radical departure from the earlier languages SL 5 has a traditional Algol like syntax with a large repertoire of control structures The success failure signaling mechanism of the earlier SNOBOL languages was extended to drive control structures in place of the more conventional use of Boolean values A notable characteristic of SL 5 is its generalized procedure mechanism [7] which provides coroutines as a natural consequence Patterns and pattern matching of the earlier languages were replaced by the concept of string scanning in which coroutine environments operate in a goal directed control regime [8] For the first time there was a mechanism for programmer-defined string scanning SL 5 also has a repertoire of elementary string processing operations that are lacking in the earlier languages The distribution of SL 5 was limited and its use at the present time is minimal

Icon represents both a synthesis of earlier ideas and a departure from trends in the earlier languages (The name Icon incidentally is not an acronym and has no special significance – although one can imagine relevant connotations)

The development of Icon as a language distinct from SL 5 was sparked by the design of a general goal-directed evaluation mechanism that allows the traditionally goal-oriented pattern matching and string scanning activities to be integrated with more conventional computational activities This integration has the effect of unifying formerly disparate features At the same time, elementary string processing operations as introduced in SL 5 have been unified with higher level string processing operations

The concept of success or failure of an operation as in the earlier languages is retained in Icon although with a slightly different interpretation Instead of operations returning a signal operations in Icon either produce a result ("succeed") or they do not produce a result ("fail") (The concept of a signal still appears in

early Icon documentation.) Some operations may generate sequences of alternative results. A goal-directed evaluation mechanism seeks alternatives from such components of an expression if other alternatives fail to produce results. In this way "trees" of alternative results in complex expressions are "searched" in the attempt to produce an overall result ("success").

Like SNOBOL4 and SL5, Icon has a variety of data types and has facilities for creating and processing structures. In many cases these facilities have been strengthened and sharpened above those of earlier languages. Icon does not have a run-time compilation facility, however.

A forewarning. Icon contains some surprises. Its goal-directed evaluation mechanism allows programming styles and techniques that other languages do not. As a consequence, learning to program in Icon is not just a matter of learning a new syntax and mastering the details of new operations — Icon allows new ways of formulating computations. The natural tendency to translate programming techniques from familiar languages to Icon may, in fact, lead to frustration. SNOBOL4 programmers, in particular, are cautioned not to blindly imitate patterns by Icon expressions of similar appearance.

1.2 Scope of the Manual

This manual describes Version 5 of the Icon programming language implemented in the C programming language [9] and designed to run under Version 7 of UNIX* [10] on PDP-11 computers.

The reader is assumed to have experience with other programming languages, a familiarity with current programming language concepts, and a working knowledge of UNIX.

This first chapter gives an overview of Icon and describes the techniques for presenting features of the language in this manual. 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.

1.3 An Overview of Icon

Icon is a general-purpose programming language with an emphasis on string processing. Icon supports a variety of data types and has facilities for creating and manipulating the commonly used kinds of structures. Storage management is automatic: there are no explicit allocation and deallocation directives. The sizes of objects are limited only by the architecture and physical limitations of the computer on which Icon runs.

Variables are "untyped" as in SNOBOL4 and SL5. Thus a variable may have values of any type. Run time type checking and coercion to expected types according to context are performed automatically.

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.

1.4 Syntax Notation

In this manual, the syntax of Icon is described in a semiformal 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 types are chosen to be mnemonic, but have no formal significance. Program text is given in a sans serif type face (e.g., write).

Alternatives are separated by bars (|). Brackets ([ ]) enclose optional items. Ellipses (...) indicate indefinite repetition of items. The metalinguistic and literal uses of bars, brackets, and periods are not mixed in any one usage, and the meaning should be clear in context. Where necessary, ambiguity is resolved by using predefined syntactic types. For example, *bar* denotes the symbol | and the symbol [is denoted by *left bracket*.

*UNIX is a trademark of Bell Laboratories.

1.5 Organization of the Manual

This manual is organized into chapters that describe the major features of the language. 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 uses of 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, some 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 specified values or optional parts of expressions. The *failure conditions* specify situations in which an operation may fail to produce a value. The *error conditions* specify situations that are erroneous and cause program termination. The defaults and error conditions are summarized in Appendices C and D.

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 and Operations

2.1 Types

Icon supports several kinds of data, called *types*

integer	procedure
real	list
string	table
cset	null
file	

Integers and real numbers (floating-point numbers) serve their conventional purposes Strings are sequences of characters as in SNOBOL4, for example 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 Lists and tables are data structures with different organizations and access methods The null value, which is represented by the symbol • in this manual, serves a special purpose as the initial value of variables The null value is illegal in most computations In addition to the types listed above, there is a facility for defining record types

The first letters of type names are used in this manual to indicate values of the corresponding types For example, i, s, and c are used to indicate integers, strings, and csets, respectively Following convention, j and k are also used to indicate integers Numerical suffixes are used when several values of the same type appear together, such as s1 and s2 The letter a ("array") is used in place of l for lists, since l is difficult to distinguish in text The letter n is used to indicate numeric types (integer or real) x and y are used to indicate objects of unspecified or undetermined type z is used for record types Where the emphasis is on expressions without regard for the values that they may produce, *expr*, *expr1*, and so on are used Liberties are taken with these conventions when the meaning is clear in context

Integers, real numbers, strings, and csets can be specified literally in the program text Integers and real numbers are represented as constants in the conventional manner For example, 300 is an integer, while 1 0 is a real number Strings are enclosed in double quotation marks, as in "summary" See Sections 4 1 1, 4 2 1, 5 2 1, and 5 3 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 Expressions

Icon is an expression-based language The most primitive expressions are identifiers and literals More complex expressions can be composed from functions, operators, control structures, and groupings The following sections describe various kinds of expressions

2.2.1 Variables and Assignment

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 an *identifier* Syntactically, an identifier must begin with a letter or underscore, which may be followed by any number of other letters, underscores, and digits Corresponding upper- and lower-case letters are distinct Reserved words, such as if, may not be used as identifiers See Appendix A for a complete list of reserved words

syntactically correct identifiers:

```
x
X
k00001
summary
Report1
node_link
_link
```

syntactically erroneous identifiers:

```
23K
report$
then
x0@s
```

There are various forms of variables other than identifiers. Some variables, such as the elements of a structure, are computed during program execution and have various syntactic representations. See Sections 6.1.4, 7.4, 8.1.2, 8.2.2, and 8.3.3.

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, v := 3 assigns the integer value 3 to the identifier v.

> *Note:* The assignment operator associates to the right and returns its left operand as a variable. Thus multiple assignments can be made. For example, v1 := v2 := 3 assigns 3 to both v1 and v2.

Any expression that yields a variable may appear on the left side of an assignment operation and any expression may appear on the right. For example, v1 := v2 assigns the value of the identifier v2 to the identifier v1.

> *Error Condition:* If the expression on the left side of the assignment operation is not a variable, Error 111 occurs.

The infix operator :=: exchanges the values of its operands. For example, v1 :=: v2 exchanges the values of v1 and v2.

> *Note:* The exchange operator associates to the right and returns its left operand as a variable.
>
> *Error Condition:* If the expression on either side of the exchange operation is not a variable, Error 111 occurs.

2.2.2 Keywords

Keywords are used to designate important values and variables. Some keywords have constants as values, others control the status of global conditions, while others provide values related to the environment in which the executing program operates.

A keyword is composed of an ampersand (&) followed by one of a number of identifiers that have special meanings. A typical keyword is &date, whose value is the current date.

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 11.3.4). 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.

> *Error Condition:* If an attempt is made to assign a value to a keyword that is not a variable, Error 111 occurs.

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

2.2.3 Functions

Functions (built-in procedures) 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 that is enclosed in parentheses:

name ([*expr* [, *expr*]])

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

As indicated, arguments may be expressions of arbitrary complexity

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

Error Condition If an argument cannot be converted to a required type, an error with a number of the form 1*nn* occurs, where *nn* identifies the expected type See Appendix D

Defaults The null value, •, is provided for omitted arguments In some cases, null values are converted to special default values This allows values that occur frequently to be omitted 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

Note If more arguments are provided in a function call than are required by the function, the extra arguments are evaluated, but their values are ignored

2.2.4 Operators

Operators provide a convenient abbreviated notation for functions There are two kinds of operators prefix and infix Example of prefix operators are -i, which produces the negative of i, and *x, which produces the size of x Examples of infix operators are i + j and i * j, which produce the sum and product of i and j, respectively

While all prefix operators are single symbols, some infix operators are composed of more than one symbol Examples are v := x, s1 || s2 (which produces the concatenation of the strings s1 and s2) and s1 == s2 (which compares strings s1 and s2 for equality)

Blanks and parentheses may be used to avoid potential ambiguities when infix operators are followed by prefix operators In the absence of blanks or parentheses, rules are used to interpret potentially ambiguous expressions See Section 12 4 In addition, rules of precedence and associativity are used to determine which operands are associated with which operators in complex expressions See Appendix A

As a class, prefix operators have the highest precedence (bind most tightly to their operands) For example, -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 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 Assignment, however, associates to the right Thus v1 := v2 := v3 is equivalent to v1 := (v2 := v3) A complete list of infix operator associativities is given in Appendix A

2.3 Evaluation of Expressions

2.3.1 Results

Some expressions produce variables The simplest example is an identifier such as delta Other expressions, such as the literal 13, produce values The term "result" is used to refer to either a variable or a value Values may be assigned to variables, and some operations, such as assignment, require operands that produce variables

Conversely, many operations require values Thus in

s1 == s2

the values of the variables s1 and s2 are compared

The process of obtaining the value of a variable is called *dereferencing* In Icon, the arguments of functions and the operands of operators are evaluated in a strictly left-to-right manner However, dereferencing is not performed by functions and operators until all arguments and operands have been evaluated Normally this does not affect the results of computation, but in cases where expressions have side effects it may Consider, for example, the expression

```
f(x, x := *x)
```

Here the second argument of f is an expression that changes the value of x. The effect is as if f(*x,*x) had been called, regardless of the original value of x, since the first argument of f is not dereferenced until the second argument has been evaluated.

Explicit dereferencing may be obtained by the prefix . operator. Thus

```
f(.x, x := *x)
```

dereferences the first argument so that evaluation of the second argument does not affect the value of the first argument.

Note. The operand of the dereferencing operator need not be a variable.

2.3.2 Success and Failure

The evaluation of an expression may either produce a result (a variable or a value), or it may fail to produce a result. Failure to produce a result may occur for a variety of reasons, but it generally indicates that some condition that is necessary for the production of a result does not hold. For example, the comparison operation i = j fails to produce a result if i is not numerically equal to j. Note that this is different from comparison in most programming languages, where the result of comparison is a Boolean value, either *true* or *false*, depending on whether or not the condition is satisfied.

In Icon, on the other hand, the course of program execution is determined by whether or not expressions produce results. For example, in the familiar control structure

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

expr2 is evaluated if *expr1* produces a result, while *expr3* is evaluated if *expr1* does not produce a result. Note that the effect of this method of control is the same as the use of Boolean values. The Icon mechanism provides more generality, however, since it allows operations to be conditional and at the same time to produce meaningful results. For example, find(s1,s2) returns the position at which s1 is a substring of s2 provided there is such a substring, but fails to produce a result if there is not such a substring.

In this manual, the term "succeeds" is used as an abbreviation for "produces a result", while "fails" is used as an abbreviation for "fails to produce a result". The term "outcome" is used to refer to the consequences of evaluating an expression, whether it be a result or failure.

Failure of expression evaluation is a normal occurrence during the course of program execution. Failure is not a programming error *per se*, but simply a way of selecting alternative paths of computation.

The keyword &fail always fails. It may be used in situations where explicit failure is desired.

2.4 Basic Control Structures

Icon provides a number of traditional control structures, as well as some that are specifically designed to utilize the failure of an expression to produce a result.

1. The control structure

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

evaluates *expr1*. If *expr1* succeeds, *expr2* is evaluated, otherwise *expr3* is evaluated. The outcome of

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

is the outcome of *expr2* or *expr3*, whichever is evaluated. If the else clause is omitted and *expr1* fails, the if-then expression fails.

2. The control structure

```
while expr1 [ do expr2 ]
```

evaluates *expr1* repeatedly until it fails. Each time *expr1* succeeds, *expr2* is evaluated. The outcome of while-do is failure, but see Section 2.6.

3 The control structure

until *expr1* [do *expr2*]

evaluates *expr1* repeatedly until it succeeds. Each time *expr1* fails, *expr2* is evaluated. The outcome of until-do is failure, but see Section 2.6.

4 The case control structure permits the selection of one of a number of expressions according to the value of a control expression. The form of the case control structure is

case *expr* of { [*case-clause* [, *case clause*]] }

where *expr* is the control expression. Case clauses have the form

expr1 : *expr2*

where *expr1* is a selector expression and *expr2* is an expression that is evaluated if *expr1* is selected. There is also a default case clause, which has the form default *expr2*. When the case expression is evaluated, the control expression is evaluated first and its value is compared to the values of the selector expressions, in order, as given in the case clauses. If a comparison is successful, the expression in the case clause is evaluated and its outcome becomes the outcome of the case expression. If no comparison succeeds, the expression in the default case clause, if present, is evaluated and becomes the outcome of the case expression.

Notes. The default clause may appear in any position with respect to the other case clauses, although it is customary for it to appear either first or last. Only one default clause is allowed in a case expression. It is evaluated as if it appeared last. The semicolons between case clauses may be omitted if the clauses are placed on separate lines.

Failure Conditions. case fails if the control expression fails, if no case clause is selected, or if the selected expression fails.

An example of a case expression is

```
case *s1 of {
   1:       m := 0
   *s2:     m := 1
   default: m := 2
   }
```

which assigns 0 to m if the size of s1 is 1, 1 to m if the size of s1 is the same as the size of s2 (but not 1), and 2 to m otherwise.

5 The control structure

repeat *expr*

evaluates *expr* repeatedly. repeat terminates only through a loop exit (see Section 2.6) or a procedure return (see Section 11.3.2).

Note. repeat has no outcome *per se*.

6 The control structure

not *expr*

produces • if *expr* fails but fails if *expr* succeeds. For example,

if not *expr1* then *expr2* else *expr3*

is equivalent to

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

2.5 Compound Expressions

Expressions may be compounded to allow a sequence of expressions to appear in a context that requires a single expression. The outcome of a compound expression is the outcome of the last expression in the sequence. A compound expression has the form

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

For example

```
if z = 0 then {x := 0; y := 1}
```

sets x to 0 and y to 1 if z is 0.

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

```
if z = 0 then {
   x := 0
   y := 1
   }
```

is equivalent to the compound expression above. See also Section 12.2.

2.6 Loop Control

There are two control structures for bypassing the normal completion of expressions in loops. These control structures may be used in **repeat**, **while-do**, **until-do**, and **every-do** (see Section 3.1).

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 *expr*

causes immediate termination of the loop without the completion of the expression in which the **break** appears. The outcome of *expr* becomes the outcome of the loop in which the **break** occurs.

Default An omitted *expr* defaults to •

2.7 Procedures

A program is composed of a sequence of declarations. Procedure declarations, which contain the executable portions of a program, have the form

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

The procedure name identifies the procedure in the same way that 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 function is called. For example

```
m := max(*s1,*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 11.

Chapter 3

Generators and Expression Evaluation

3.1 Generators

Some expressions, called *generators*, are capable of producing a sequence of results An example is find(s1,s2), which produces the positions at which s1 occurs as a substring of s2 For example, in

```
find("th","this is the thesis")
```

there are three positions at which th occurs as a substring of the second argument 1, 9, and 13 On the other hand, in

```
find("th","a single thesis")
```

there is only one position, 10 In fact, there may be no position, as in

```
find("th","we have none")
```

In this case, find cannot produce a result (it fails) Note that the number of results that a generator like find can produce depends on the values of its arguments

If a generator is used in a simple computational context, it produces only its first result For example

```
i = find("th","this is the thesis")
```

assigns the value 1 to i On the other hand, in

```
i = find("th","we have none")
```

the function find fails and the value of i is not changed

There are a number of contexts in which some or all of the results produced by a generator may be useful The control structure

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

evaluates *expr2* for every result produced by *expr1* For example

```
every i = find(s1,s2) do write(i)
```

writes all the positions at which s1 occurs as a substring of s2

Note The outcome of every-do itself is failure

As indicated, the do clause is optional The example above can be written more concisely as

```
every write(find(s1,s2))
```

Note that although write is not a generator itself, write is called for every value of its argument, which is a generator The same situation occurs in the assignment operation above

There are a number of generators Two of the most fundamental generators are alternation and integer sequencing

Alternation is a control structure that has the form

```
expr1 | expr2
```

This control structure produces the sequence of results produced by *expr1* followed by the sequence of results produced by *expr2* For example

```
every f(1 | 3)
```

evaluates f(1) and f(3)

In this case both the expressions in alternation are simple values *expr1* and *expr2* may be generators in which case each produces its sequence For example,

```
every write(find(s1,s2) | find(s3,s4))
```

writes all the positions at which s1 occurs as a substring of s2 followed by all the positions at which s3 occurs as a substring of s4 Similarly

```
every write(find(s1 | s2,s3))
```

writes all the positions at which s1 occurs as a substring of s3 followed by all the positions at which s2 occurs as a substring of s3

The operation

expr1 to *expr2* [by *expr3*]

generates the integers in sequence from the value of *expr1* to the value of *expr2*, inclusive, using the value of *expr3* as an increment For example

```
every write(0 to 10 by 2)
```

writes 0, 2, 4, 6 8, and 10

Error Condition If the value produced by *expr3* is 0 Error 211 occurs

Notes *expr1* *expr2* and *expr3* are evaluated only once Generation stops when *expr2* is exceeded *expr3* may be negative in which case successively smaller values are generated until *expr2* is reached or passed The construction every i := j to k do *expr* is similar to the for control structure found in many programming languages

Default If the by clause is omitted the increment defaults to 1

In some cases it is desirable to limit the number of results produced by a sequence The control structure

expr1 \ *expr2*

produces at most *expr2* results from the sequence generated by *expr1* For example

```
every write(find(s1,s2) \ 10)
```

writes the first 10 positions at which s1 occurs as a substring of s2 Of course if there are fewer than 10 positions only those values are produced

Sometimes it is useful to repeatedly produce the sequence produced by a generator The repeated alternation control structure

|*expr*

is equivalent to

expr | *expr* | *expr*

except that if *expr* ever fails (that is, if it ever produces no result), |*expr* terminates (This may occur because *expr* fails initially or because of side effects that affect the sequence produced by *expr*) For example

```
every write( |find(s1,s2) \ 100)
```

writes the positions at which s1 occurs as a substring of s2 repeatedly, but terminates after 100 values have been written

3.2 Goal-Directed Evaluation

In every *expr1* do *expr2* the complete sequence of results of *expr1* is produced by the explicit use of every Expressions in Icon are evaluated in a *goal-directed* fashion in which the results of generators are automatically produced in sequence if that is necessary for an enclosing expression to succeed (that is to produce a result) A simple example of goal-directed evaluation is illustrated by

```
(x | y) > 0
```

Here the left operand of the comparison operator is a generator capable of producing two results the variables x and y The value of the first result of the alternation is compared to 0 If this comparison succeeds the entire expression succeeds However, if this comparison fails, the entire expression does not necessarily fail Instead, the second result of the alternation is produced and is compared to 0 Hence the entire expression succeeds if the value of either x or y is greater than 0 (hence the term "alternation")

This goal-directed evaluation mechanism is completely general and applies to the evaluation of the arguments and operands of all functions and operators For example

```
if (x | y | z) > (a | b) then write("plateau reached")
```

writes plateau reached if any of x y, or z is greater than either a or b

3.3 Evaluation of Expressions

The arguments of functions and procedures, as well as the operands of operators, are evaluated from left to right In goal-directed evaluation, if evaluation of an argument or operand fails to produce a result, *control backtracking* takes place to the most recently evaluated argument or operand to obtain another result from its sequence For example in

expr1 + *expr2*

expr1 is evaluated first If *expr1* fails, the addition operation fails If *expr1* succeeds, *expr2* is evaluated If *expr2* fails however, the addition operation does not necessarily fail Instead, backtracking occurs and another result from the sequence for *expr1* is sought If such an alternative result exists, *expr2* is evaluated again Since the evaluation of *expr1* may affect *expr2* (by means of side effects), *expr2* may now succeed If so, the addition is performed An example of such a situation is

```
(x := n to m) + find("1",x)
```

In the case of a function call such as f(*expr1*,*expr2*), if *expr2* fails, alternative results are sought for *expr1* In fact if *expr1* and *expr2* both succeed but the function itself fails, alternatives are sought for the arguments (first *expr2* and failing that *expr1*) If any argument has an alternative, successive arguments are re-evaluated and the function is called again If the function continues to fail, it is called for all alternative values of the arguments The overall expression fails only if the function fails for all alternative values of the arguments This method of evaluation applies regardless of the number of arguments in the function call Operands of operators are evaluated in the same was as arguments of functions

In some cases backtracking to achieve mutual results from two expressions may be desired, even though no computation is to be performed on the results The infix operator & ("conjunction") behaves like any other infix operator with respect to backtracking, except that if *expr1* succeeds the outcome of *expr1* & *expr2* is simply the outcome of *expr2*

If mutual evaluation among several expressions is needed conjunction can be compounded, as in

expr1 & *expr2* & & *exprn*

This notation becomes cumbersome especially if the expressions are themselves complex Such compounded conjunctions may be difficult to compose correctly and to understand An alternative method is *mutual evaluation*, denoted by

(*expr1*,*expr2*, ,*exprn*)

which evaluates *expr1* *expr2* *exprn* just like the arguments in a function call If all the expressions produce results the result of mutual evaluation is the result of *exprn* Otherwise, it fails The effect is exactly the same as in a compound conjunction

Sometimes a number of expressions need to be mutually evaluated, but a result other than the last is desired The expression

expr(*expr1*,*expr2*, ,*exprn*)

produces the result of *expri* where the value of *expr* is the integer *i*, provided all the expressions produce a

result If any expression fails, however, the mutual conjunction fails For example, the value of

```
2(find(s1,s),find(s2,s),find(s3,s))
```

is the position of s2 as a substring of s provided s1 s2, and s3 all occur as substrings of s The value of *expr* can be negative, in which case the result is selected from right to left This method of selecting the result of mutual evaluation makes it easy to select the last result from a long list For example

```
(−1)(expr1, expr2,    , exprn)
```

selects the result of *exprn* The parentheses around −1 are necessary, the expression

```
−1(expr1, expr2,    , exprn)
```

produces the negative of the result of *expr1*

Note that mutual conjunction has the same syntax as a function call There is no ambiguity, however If the value of *expr* is an integer *i* the result of is the result of *expri* If the value of *expr* is a function, however, the function is applied to the arguments and the outcome is the outcome of the function call

3.4 The Extent of Backtracking

Backtracking is limited in its extent by syntactic constructions in the program The extent of backtracking therefore can be determined by examination of the text of the program (that is the extent of backtracking is not determined by the history of computation in the program)

In addition to the control structure *expr1* \ *expr2* that is described in Section 3 1 several constructions specifically limit the extent of backtracking The semicolons that separate expressions in a sequence for example, prevent backtracking from occurring between the expressions For example, in the sequence

```
expr1, expr2
```

failure of *expr2* does not cause backtracking into *expr1*

The other contexts in which goal directed evaluation is implicitly limited to one result are the first (control) expressions in the case-of, if-then-else not, until-do, and while-do

3.5 The Reversal of Effects

As described above control backtracking to an earlier point in a computation may take place in order to obtain alternative results of generators There is, however, no implicit reversal of effects such as assignments For example, in the expression

```
(y  = 1 to 10) & (y > z)
```

if the value of z is 20 the value of y after the failure of the conjunction is 10 regardless of what the value of y was before evaluation of the conjunction

There are two assignment operators that do reverse their effects if failure occurs

1 The infix operator v <− x assigns the value of x to v but restores the previous value of v if backtracking causes failure in the expression in which the reversible assignment occurred For example in

```
y  = 0, (y <− 1 to 10) & (y > z)
```

if the value of z is 20 the value of y is restored to 0 when the conjunction fails

2 The infix operator v1 <−> v2 exchanges the values of v1 and v2, but restores their former values if backtracking causes failure in the expression in which the reversible exchange occurred

Notes The reversible assignment and exchange operators associate to the right and return their left operands as variables

Error Conditions If the expression on the left side of the reversible assignment operation or either expression in the reversible exchange operation is not a variable Error 111 occurs

Chapter 4

Numbers and Arithmetic Operations

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

4.1 Integers

Integers in Icon are treated as they are in most programming languages

Note The allowable range of integer values is -2^{31} to $2^{31}-1$

4.1.1 Literal Integers

Integers may be specified literally in a program in the conventional fashion

Notes Leading zeroes are allowed but are ignored Negative integers cannot be expressed literally but they may be computed as the results of arithmetic operations

Examples

expression	*value*
0	0
000	0
10	10
010	10
27524	27 524

Integer literals such as those given above are in the base 10 Other radices may be specified by beginning the integer literal with *n*r, 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

Notes The digits used in the literal must be less than the radix Either r or R may be used to indicate a radix literal Either upper- or lower case letters may be used for "digits"

Examples

expression	*value*
2r11	3
8r10	8
10r10	10
16rff	255
16RFF	255
36rCat	15 941

4.1.2 Integer Arithmetic

The following infix arithmetic operations are provided

expression	*operation*	*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	remaindering	2	left
i ^ j	exponentiation	3	right

Notes The remainder of integer division is discarded, that is, the result is truncated. i % j produces the remainder of i divided by j. The sign of the result is the sign of i.

Error Conditions If an attempt is made to divide by 0, Error 201 occurs. If the second operand of remaindering is zero, Error 202 occurs. If the result of an arithmetic operation exceeds the range of allowable integer values, Error 203 occurs.

Examples

expression	*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	262,144
4 % 3	1
1400 % 1000	400
4 % 4	0
-4 % 3	-1
4 % -3	1
-4 % -3	-1

There are two arithmetic prefix operations: +i and -i, to form the positive and negative of i respectively. In addition, the function **abs(i)** produces the absolute value of i.

Examples

expression	*value*
+100	100
-100	-100
+0	0
-0	0
-(4 - 700)	696
abs(7)	7
abs(-7)	7

4.1.3 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 return the value of their right operand if the specified relation between the operands holds and fail otherwise.

Examples

expression	*value*
100 100	100
1 ~= 1	*none*
1 > 1	*none*
2 > 1	1
1 < 2	2
2 >= 1	1
2 <= 2	2
2 < 3 < 400	400
2 < 3 = 4	*none*

4.2 Real Numbers

Real numbers are represented in floating-point format

Note Floating-point numbers are double precision

4.2.1 Literal Real Numbers

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

Notes For magnitudes less than 1, a leading zero is required Additional leading zeroes are allowed but are ignored Either e or E may be used in exponent notation

Examples

expression	*value*
3 14159	3 14159
0 0	0 0
000	0 0
27e2	2,700 0
27e 6	0 000027
27e5	2,700,000 0
27E5	2,700,000 0

4.2.2 Real Arithmetic

The arithmetic operations available for real numbers are the same as those available for integers See Section 4 1 2

Error Conditions In the case of real overflow, real underflow, or division by zero, Error 204 occurs If an attempt is made to raise a negative real number to a real power, Error 206 occurs

Examples.

expression	*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
4 7 % 2 0	0.7
2.5 % 1 0	0.5
+1.0	1.0
-1 0	-1.0
abs(7 0)	7.0
abs(-7.0)	7.0

4.2.3 Comparison of Real Numbers

The comparison operations available for real numbers are the same as those available for integers. See Section 4.1.3.

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

expression	*value*
1.0 = 1.0	1 0
1 0 ~= 1.0	*none*
1 0 > 1 0	*none*
2.0 > 1 0	1.0
1 0 < 2 0	2.0
2.0 <= 1.0	*none*
2 0 <= 2 0	2.0
2 0 < 3 0 < 4 0	4.0
2 0 < 3.0 <= 4.0	4.0
2 0 < 3 0 = 4 0	*none*

4.3 Mixed-Mode Arithmetic

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

Examples

expression	*value*
10 + 2	30
1 + 20	30
1 20	10
10 * 2	20
10 / 2	05
2 / 10	20
1 - 1 - 10	-10
1 * 20 / 2	10
1 / 20 * 2	10
10 / 2 * 2	10
20 ^ 2	40
20 ^ -1	05

4.4 Arithmetic Type Conversion

4.4.1 Conversion to Integer

The value of integer(x) is an integer corresponding to x where x may be an integer, real number or cset

1 Integers are returned unmodified by integer(x)

2 Real numbers are converted to integer by truncation

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

Examples

expression	*value*
integer(2.0)	2
integer(2.5)	2
integer(-2.5)	-2
integer(2e35)	*none*

3 Strings are converted to integers in the same way that an integer literal is treated in program text except that

(a) Leading and trailing blanks are allowed but are ignored

(b) A leading sign may be included

(c) There must be at least one digit

If the string corresponds to a real literal, real-to-integer conversion is performed See Section 5.4

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

Examples

expression	*value*
integer('10")	10
integer('8r10")	8
integer("-10")	-10
integer(3)	3
integer(" 0003")	3
integer("3.5")	3
integer('3 x")	*none*
integer('3r4 ')	*none*

4 Csets are first converted to strings and then to integers See Section 5 4

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

For operations that require integers, implicit conversions are automatically performed for real numbers, strings and csets

Error Condition If an implicit conversion to integer fails Error 102 occurs

Examples

expression	*value*
1 + "10"	11
'2' ^ 4 0	16 0

4.4.2 Conversion to Real Number

The value of real(x) is a real number corresponding to x, where x may be a real number, integer, string, or cset

1 Real numbers are returned unmodified by real(x)

2 Integers are converted to the corresponding real values

Examples

expression	*value*
real(10)	10 0
real(−10)	−10 0
real(8r10)	8 0
real(27000)	27 000 0

3 Strings are converted to real numbers in the same way that real literals are treated in program text except that

(a) Leading and trailing blanks are allowed but they are ignored

(b) A leading sign may be included

(c) 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

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

Examples

expression	*value*
real("10 0")	10 0
real("−10 0")	−10 0
real("27000")	27 000 0
real(" 3 0")	3 0
real(" 0003 0")	3 0
real("8r10")	8 0
real("3 x")	*none*
real("3r4")	*none*

4 Csets are first converted to strings and then to real numbers See Section 5 4

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

Implicit conversions are automatically performed for integers strings, and csets in operations that require real numbers

Error Condition If an implicit conversion to real number fails Error 102 occurs

Examples

expression	*value*
1 0 + "10 0"	11 0
"2 0" ^ 3	8 0

4.5 Conversion to Numeric

The function numeric(n) returns the integer or real number corresponding to n if n is an integer, real number, or if it is convertible to one of these types See Section 5 4 The function fails otherwise

Examples

expression	*value*
numeric(100)	100
numeric(0 0)	0 0
numeric("0")	0
numeric("0 0")	0 0
numeric("a")	*none*
numeric("16Rff")	255
numeric("3r4")	*none*
numeric(" ")	*none*

Chapter 5

Strings and Character Sets

5.1 Characters

Although characters are not themselves data objects in Icon, strings of characters and sets of characters are. Strings form the heart of Icon's processing capabilities.

The character set used by Icon is based on ASCII [11], There are, however, 256 different characters available for use in Icon programs.

Note: The thirty-third character (octal code 40) is the blank (space). Since it has no visible representation, the symbol □ is used in this manual to represent the blank in contexts that otherwise might be confusing.

While it is customary to think of characters in terms of their graphic representations and control functions, characters are basically just integers. Internally the integers corresponding to ASCII are represented by octal codes from 000 through 177 (hexadecimal codes 00 through 7F). 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 6.2) and for sorting (see Section 8.4). The full set of 256 characters similarly are represented by octal codes 000 through 377 (hexadecimal codes 00 through FF).

5.2 Strings

A string is a sequence of zero or more characters. Any character may appear in a string. There are many ways of constructing strings during program execution. See Chapter 6.

5.2.1 Literal Strings

Strings may be specified literally in a program by delimiting (enclosing) the sequence of characters by double quotes (").

Examples:

expression	*value*
"X"	X
"□"	□
"abcd"	abcd
"Isn't□it□great?"	Isn't□it□great?

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

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

character	*code*
backspace	\b
delete	\d
escape	\e
formfeed	\f
linefeed	\l
newline	\n
carriage return	\r
horizontal tab	\t
vertical tab	\v
double quote	\"
single quote	\'
backslash	\\
octal code	*ddd*
hexadecimal code	\x*dd*
control code	\^*c*

The specification *ddd* represents the character with octal code *ddd*. The specification \x*dd* represents a character with hexadecimal code *dd*. Only enough digits need to be given to specify the octal or hexadecimal code. For example, \0 specifies the null character and \xa is equivalent to \x0a. \^*c* represents the ASCII character control-*c*. For example, \^A is the ASCII character control-A. In general, \^*c* is the character corresponding to the five low-order bits of *c*. If the character following a backslash is not one of those listed above, the backslash is ignored.

Notes. The convention used here for representing characters in literals is adapted from that used by the C programming language [9]. The linefeed and newline characters are the same.

Examples

expression	*value*
"\"oops\" "	"oops"
"\"\" "	" "
"\□"	□
"\a\z"	az
"\132"	Z
"\134\134"	\\
"\77a"	?a
"\1234"	S4
"\x64"	d
"\\"	\

5.2.2 String Size

The size of a string is the number of characters it contains and is computed by the unary operator *. The empty string is the string consisting of no characters and has size zero. It may be represented literally by two adjacent quotes, enclosing no characters.

Notes: The maximum size of a string is $2^{15}-1$. The practical maximum is usually dictated by the amount of memory available. Since the empty string contains no characters, it has no visible representation. In this manual, the symbol ■ is used to represent the empty string in contexts that otherwise might be confusing. Thus "" and ■ both indicate an empty string.

Examples

expression	*value*
*"abcd"	4
*"□"	1
*""	0

5.3 Character Sets

Whereas a string is an ordered sequence of characters in which the same character may appear more than once, a character set (cset) is an unordered collection of characters The value of the keyword **&cset** is the set of all 256 characters Other character sets are subsets of **&cset** and are useful for operations where specific characters are of interest, regardless of the order in which they appear See Sections 6 3 2 and 7.3 Other built-in character sets are **&ascii**, the first 128 characters of **&cset**, **&lcase**, the lower-case letters, and **&ucase**, the upper-case letters

Error Condition The keywords **&cset**, **&ascii**, **&lcase**, and **&ucase** are not variables If an attempt is made to assign a value to one of them, Error 111 occurs

Csets may be specified literally in a program by delimiting (enclosing) the characters in single quotes (') Duplicate characters in cset literals are ignored and the order of the characters is irrelevant The same escape conventions that apply to string literals apply to character set literals

Examples

expression	*value*
'abcd'	a b c d
'badc'	a b c d
'energy'	e g n r y
'\\'	\

Note Values of csets in examples are given with separating spaces to distinguish them from the values of strings

There are five operations on character sets

1 ~c is the complement of c with respect to **&cset**

2 c1 ++ c2 is the union of c1 and c2

3 c1 ** c2 is the intersection of c1 and c2

4 c1 -- c2 is the difference of c1 and c2, that is, all of the characters in c1 that are not in c2

5 *c is the number of characters in c

Examples

expression	*value*
c1 := 'drama'	a d m r
c2 := 'append'	a d e n p
c1 ++ c2	a d e m n p r
c1 ** c2	a d
c1 -- c2	m r
c1 -- ~c2	a d
*c1	4

Note A character set may be empty, i e containing no characters Such a character set may be obtained by '' or ~&cset

5.4 Type Conversion

5.4.1 Explicit Conversion

The value of string(x) is a string corresponding to x, where x may be an string, integer, real number, or cset

1. Strings are returned unmodified by string(x).
2. For integers and real numbers, 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.

Examples:

expression	*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 csets, the result is a string of characters in the cset, arranged in order of collating sequence (see Section 6.2).

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

The value of cset(x) is a character set corresponding to x, where x may be an integer, real number, string, or cset. If x is an integer or real number, it is first converted to a string as described above.

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

Examples:

expression	*value*
cset("drama")	a d m r
cset(1088)	0 1 8
cset(3.14)	. 1 3 4

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

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

eliminates duplicate characters and sorts the characters of the string.

Examples:

expression	*value*
string(cset("ab"))	ab
string(cset("ba"))	ab
string(cset("mam"))	am
string(cset("a□b"))	□ab

5.4.2 Implicit Conversion

In contexts that require strings, implicit conversion is automatically performed for integers, real numbers, and csets.

Error Condition: If an object of any other type is encountered in a context that requires implicit conversion to a string, Error 103 occurs.

Examples:

expression	*value*
*10	2
*010	2
*100	3

For operations that require csets, implicit conversion is performed automatically for integers, real numbers, and strings. Integers and real numbers are first converted to strings and then to csets.

Error Condition If an object of any other type is encountered in a context that requires implicit conversion to a cset, Error 104 occurs.

Chapter 6

Basic String Operations

6.1 Constructing Strings

There are a number of operations for constructing strings. Most of these operations are described in the following sections. See also Sections 7.2, 7.3, and 7.4.

6.1.1 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 empty string is the identity with respect to concatenation. That is, the result of concatenating the empty string with any string **s** is simply **s**

Examples

expression	*value*
"a" || "z"	az
"[" || "abcd" || "]"	[abcd]
"abcd" || ""	abcd
"" || ""	■

6.1.2 String Replication

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

Error Condition. If i is negative or greater than $2^{15}-1$, Error 205 occurs.

Note The value of **repl(s,0)** is ■.

Examples

expression	*value*
repl("a",2)	aa
repl("* ",3)	* *.*
repl(&lcase,0)	■

6.1.3 Positioning Strings

Positioning data in strings of a specified size is frequently useful, especially when printing output in columns. 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 value for **s2** defaults to □

Error Condition If i is negative or greater than $2^{15}-1$, Error 205 occurs

Examples

expression	*value*
left("abcd",6," □")	abcd □
left("abcd",7," □")	abcd□ □
left("abcde",7," □")	abcde □
left("abcd",6)	abcd□□
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. If the size of s1 is greater than i, it is truncated at the left end

Default A null value for s2 defaults to □

Error Condition If i is negative or greater than $2^{15}-1$, Error 205 occurs

Examples

expression	*value*
right("abcd",6," □")	□abcd
right("abcd",7," □")	□ abcd
right("abcde",7," □")	□abcde
right("abcd",6)	□□abcd
right(&lcase,10)	qrstuvwxyz

3 The value of center(s1,i,s2) is s1 centered in a string of size i s2 is used for filling on the left and right as for the functions above If the size of s1 is greater than i, it is truncated at the left and at the right to produce its center section If s1 cannot be centered exactly, it is positioned to the left of center

Default A null value for s2 defaults to □

Error Condition If i is negative or greater than $2^{15}-1$, Error 205 occurs

Examples

expression	*value*
center("abcd",8," □")	□abcd □
center("abcd",9," □")	□abcd□ □
center("abcde",9," □")	□abcde □
center("abcd",6)	□abcd□
center(&lcase,10)	ijklmnopqr
center(&lcase,11)	ijklmnopqrs

6.1.4 Character Positions and Substrings

The positions of characters in a string are numbered from the left starting at 1 The numbering identifies positions between characters For example, the positions in the string HAT are

```
 H  A  T
↑  ↑  ↑  ↑
1  2  3  4
```

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

```
 H  A  T
↑  ↑  ↑  ↑
3  2  1  0
```

For this string, positions 4 and 0 are equivalent, positions 3 and −1 are equivalent, and so on

The positions that can be specified for a string s are in the range −*s to *s + 1, inclusive Values out of this range are not allowable position specifications In general, the positive specification i is equivalent to the nonpositive specification i−(*s + 1)

Note The only allowable positions for the empty string are 1 and 0, which are equivalent

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 Substrings are determined by beginning and ending positions, using a *range specification* There are four forms of range specification

i	the single character following position i
i:j	characters between positions i and j
i+ k	k characters following position i
i− k	k characters preceding position i

In all cases, i and j may be given by positive or nonpositive specifications and k may be positive, negative, or zero

Note The range specifications i j and j i are equivalent

A substring is obtained by a subscripting expression of the form

string left-bracket range-specification right-bracket

The resulting substring consists of the characters given by the range specification

Failure Condition A subscripting expression fails if either of the positions of the range specification do not correspond to allowable positions in the string being subscripted In this case, the specification is said to be *out of range*

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 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

Examples

expression	*value*
&lcase[1]	a
&ucase[26]	Z
&lcase[1 2]	a
&lcase[2 1]	a
&lcase[1 1]	■
&ucase[27]	*none*
&lcase[27 28]	*none*
&lcase[1 −2]	y
"abcd"[2 0]	bcd
"abcd"[2 7]	*none*
"abcd"[1 0]	abcd
"abcd"[2+ 2]	bc
"abcd"[3-- 2]	ab

If the string specified in a substring operation is a variable, assignment can be performed to replace the specified substring and hence change the value of the variable

Notes All forms of assignment can be used to replace substrings

Error Condition If an attempt is made to assign to a subscripting expression in which the string is not a variable Error 111 occurs

Examples

expression	*value of* s
s = "abcd"	abcd
s[1 2] = "xx"	xxbcd
s[-1 0] = ""	xxbc
s[1] ~ "abc"	abcxbc
s[1+2] = "y"	ycxbc
s[2] ~ s[3]	yxcbc

6.1.5 Other String-Valued Operations

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

Examples

expression	*value*
reverse("abcd")	dcba
reverse(&lcase)	zyxwvutsrqponmlkjihgfedcba
reverse("")	■

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 A null value for c defaults to '□'

Examples

expression	*value*
trim("abcd□□□",'□')	abcd
trim("abcd□□□")	abcd
trim("abcd□□□",'□d')	abc
trim("abcd□□□",'d')	abcd□□□
trim("abcd□□□",&ascii)	■

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 If the sizes of s2 and s3 are not the same, Error 208 occurs

Defaults A null value for s2 defaults to &ucase and a null value for s3 defaults to &lcase

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

Examples

expression	*value*
map("abcda","a","*")	*bcd*
map("abcda","ad","**")	*bc**
map("abcda","ad","* ")	*bc *
map("abcda","ax","* ")	*bcd*
map("abcda","yx","* ")	abcda
map("abcd","bcad","1234")	3124
map("acda","aac","123")	23d2
map("wxyz" "zyxw" "abcd")	dcba

6.2 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 and is based on the collating sequence If a character c1 appears before c2 in collating sequence, 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 empty string is lexically less than any other string

The operation s1 << s2 succeeds if s1 is lexically less than s2 and fails otherwise The value returned on success is s2 In all, there are six lexical comparison operators

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

Examples

expression	*value*
"X" << "x"	x
"x" <<= "X"	*none*
"x" >> "x"	*none*
"XX" << "x"	x
"xx" >>= "xX"	xX
"xx" << "xxx"	xxx
"xx" << "xxX"	xxX
"" ~== "x"	x
"" == ""	■

6.3 String Analysis

Most programming operations on strings involve analysis rather than synthesis, and the repertoire of analytic operations is correspondingly large

6.3.1 Identifying Substrings

There are two functions for identifying specific substrings

1 If s1 is an initial substring of s2[i j], the function match(s1,s2,i,j) returns the position of the end of the substring

Failure Condition match(s1,s2,i,j) fails if s1 is not an initial substring of s2[i j]

Defaults A null value for i defaults to 1 and a null value for j defaults to 0

Examples

expression	*value*
match("a","abc",1)	2
match("a","abc")	2
match("a","abc",2)	*none*
match("ab","abc",1,2)	*none*
match("bc","abc",1)	*none*
match("bc","abc",2)	4
match("bcd","abc",2)	*none*
match(" ","abcd",1)	1
match(" ","abcd",5)	5

2 The value of find(s1,s2,i,j) is the leftmost position in s2 where s1 occurs as a substring in s2[i:j]

Failure Condition find(s1,s2,i,j) fails if s1 is not a substring of s2[i:j]

Defaults A null value for i defaults to 1 and a null value for j defaults to 0

Examples

expression	*value*
find("a","abcd",1)	1
find("a","abcd")	1
find("bc","abcd",1)	2
find("a","abcd",2)	*none*
find("ab","abcd",1,2)	*none*
find("de","abcd",1)	*none*
find(" " "abcd",3)	3

The function find is a generator that produces the sequence of the positions, from left to right, at which s1 is a substring of s2[i:j]

Examples

expression	*values in sequence*
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

6.3.2 Lexical Analysis

Lexical analysis involves sets of characters rather than substrings There are four lexical analysis functions

1 If the first character of s[i:j] is contained in the character set c, the value of any(c,s,i,j) is i+1

Failure Condition any(c,s,i,j) fails if the first character of s[i:j] is not contained in the character set c

Defaults A null value for i defaults to 1 and a null value for j defaults to 0

Examples

expression	*value*
any('abc',"abcd",1)	2
any('abc',"abcd")	2
any('abc',"dcba")	*none*
any(~'abc',"dcba")	2
any('abc',"dcba",2)	3
any('abcd',"abcd",1,2)	2

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

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

Defaults: A null value for i defaults to 1 and a null value for j defaults to 0.

Examples

expression	*value*
upto('a',"abcd",1)	1
upto('a',"abcd")	1
upto('abc',"abcd")	1
upto(~'abc',"abcd")	4
upto('d',"abcd",2)	4
upto('d',"abcd",2,3)	*none*
upto('a',"abcd",2)	*none*

The function upto is a generator that produces the sequence of the positions, from left to right, at which a character of c occurs in s[i:j].

Examples

expression	*values in sequence*
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 s[i:j] consisting solely of characters contained in c

Failure Condition many(c,s,i,j) fails if the first character of s[i:j] is not contained in c

Defaults: A null value for i defaults to 1 and a null value for j defaults to 0.

Examples

expression	*value*
many('ab',"abcd",1)	3
many('ab',"abcd")	3
many('ab',"abcd",2)	3
many('ab',"abcd",2,3)	3
many('ab',"abcd",3)	*none*

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

In determining balance, a count is kept, starting at 0. Characters in s[i:j] are processed from left to right If the character being processed is contained in c1 and the count is zero, the process is complete at that point Otherwise, 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.

Failure Conditions If the count ever becomes negative or if the substring being examined is exhausted with a positive count, bal fails

Note Characters in c2 are examined before characters in c3, so that if a character occurs in both c2 and c3, it is treated as if it occurred only in c2

Defaults A null value for i defaults to 1 and a null value for j defaults to 0 A null value for c1 defaults to &cset, a null value for c2 defaults to '(', and a null value for c3 defaults to ')'

Examples

expression	*value*
bal('+','(',')',"(a)+(b)")	4
bal('+',,,"(a)+(b)",1)	4
bal('+',,,"(a)+(b)")	4
bal('+',,,"(a)+(b)",2)	*none*
bal('-',,,"(a)+(b)")	*none*
bal(,,,"(a)+(b)")	1
bal(,'([',']')',"(a)+(b)')	1

The function bal is a generator that produces the sequence of positions, from left to right, at which successively longer balanced strings terminate

Examples

expression	*values in sequence*
every bal(,,,"(a)+(b)+(c)")	1, 4, 5, 8, 9
every bal('+',,,"(a)+(b)+(c)")	4, 8
every bal(,,,"abcd")	1, 2, 3, 4

Chapter 7

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

The string scanning expression

expr1 ? *expr2*

evaluates *expr1* and establishes its value as the string to be scanned *expr2* is then evaluated to perform the scanning The outcome of the string scanning expression is the outcome of *expr2*

7.1 Scanning Keywords

During string scanning, the string being scanned is the value of the keyword &subject The implicit position in &subject is the value of the keyword &pos The value of &subject is automatically set to the value of *expr1* and the value of &pos is set to 1, corresponding to the beginning of &subject Subsequently, values may be explicitly assigned to &subject and &pos Assignment of a value to &subject automatically sets &pos to 1, as does assignment to a substring of &subject

Note A nonpositive position specification may be used in assignment to &pos, but the corresponding positive value is actually assigned

Failure Condition An attempt to set &pos to a value that is out of the range of &subject fails

The function pos(i) returns the positive equivalent of the position i in &subject, provided &pos is at this position

Failure Condition pos(i) fails if &pos is not at position i

Examples

expression	*value*	*value of* &pos
&subject = "abcd"	abcd	1
pos(1)	1	1
pos(−4)	1	1
pos(3)	*none*	1
&pos = −1	4	4
pos(−1)	4	4
&subject[2 4] = 'x"	x	3
&subject = "ab"	ab	1

7.2 Positional Analysis

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

i The result of move(i) is the substring between &pos and &pos+i, and &pos is incremented by i

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

Examples:

expression	*value*	*value of* &pos
&subject := "abcd"	abcd	1
move(2)	ab	3
move(3)	*none*	3
move(-1)	b	2
move(-2)	*none*	2
move(0)	■	2
&pos := 0	5	5
move(-1)	d	4

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:

expression	*value*	*value of* &pos
&subject := "abcd"	abcd	1
move(2) & move(3)	*none*	1
move(2)	ab	3
move(-1) & pos(3)	*none*	3

2. The result 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:

expression	*value*	*value of* &pos
&subject := "abcd"	abcd	1
tab(2)	a	2
tab(0)	bcd	5
tab(1)	abcd	1
tab(-5)	*none*	1

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

Examples:

expression	*value*	*value of* &pos
&subject := "abcd"	abcd	1
tab(0) & move(1)	*none*	1
tab(0) & move(-1)	d	4

7.3 Scanning Operations

Several functions have defaults that provide implicit arguments for string scanning:

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)
many(c)	many(c,&subject,&pos,0)
match(s)	match(s,&subject,&pos,0)
upto(c)	upto(c,&subject,&pos,0)

Thus in each case the default interpretation 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.

Note The default interpretations for the last two arguments apply only if the argument that specifies the string to be examined is omitted or •. See Appendix C.

Examples

expression	*value*	*value of* &pos
&subject := "abcd"	abcd	1
upto('c')	3	1
upto('a')	1	1
many('abc')	4	1
any('d')	*none*	1

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

Examples

expression	*value*	*value of* &pos
&subject := "abcd"	abcd	1
tab(upto('c'))	ab	3
tab(upto('a'))	*none*	3
tab(many('c'))	c	4
tab(any('d'))	d	5

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

Examples

expression	*value*	*value of* &pos
&subject := "abcd"	abcd	1
="ab"	ab	3
="ab"	*none*	3
="c"	c	4
="d"	d	5
=" "	■	5
='d"	*none*	5

7.4 Nested Scanning

The values of &subject and &pos are saved on entry to string scanning and are restored upon exit. Consequently, nested scanning is possible. For example, suppose words contains a sequence of words followed by blanks. Then the following code segment assigns a similar string to twords, but with only those words containing a t. It also assigns the total number of words to wcount.

```
twords := ""
wcount := 0
words ?
   while tab(upto('□')) ? {
      if upto('t') then twords := twords || &subject || "□"
      wcount := wcount + 1
      }
   do move(1)
```

Warning The values of &subject and &pos are not restored if string scanning is exited by a break, next or a procedure return.

7.5 Generation During Scanning

Like in any other operation, both operands in string scanning can be generators. For example,

```
every write((s1 | s2 | s3) ? upto(c1 | c2))
```

writes every position at which a character of c1 or c2 occurs in s1, s2, and s3. The order in which the values are produced is not the same as in

```
upto(c1 | c2, s1 | s2 | s3)
```

since the order in which the arguments are reactivated to produce alternatives is different.

Chapter 8

Structures

Structures are aggregates of variables. Different kinds of structures have different organizations and different methods for accessing these variables. Structures are data objects and may be assigned to variables like other data objects. Structures are not copied when they are assigned to variables.

Note. There are specific limits to the sizes of structures as noted in subsequent sections. In practice maximum sizes are usually limited by the amount of available memory.

8.1 Lists

Lists are sequences of variables that can be referenced by position or manipulated by stack and queue access methods. When referenced by positions, lists appear to be one-dimensional arrays. When manipulated by stack and queue access methods, lists expand and contract as needed. Positional and access methods for lists can be used in combination.

8.1.1 Creation of Lists

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

left-bracket expr [, *expr*] *right-bracket*

where the values of the expressions provide the initial values of the list elements.

The value of *a gives the number of elements in a.

Note. The value of [ ] is an empty list containing no elements. In other cases, omitted arguments default to the •. For example, [ ] is a list of two null-valued elements.

Examples

expression	*value*
triple := [0,0 0]	[0 0 0]
*triple	3
line := [,]	[• • • •]
*line	4
seq := [1,2 3 4 5 6 7 8]	[1 2 3 4 5 6 7 8]
*seq	8
unit := [ ]	[ ]
*unit	0

Lists are also created by the function list(i,x) where i is the size of the list and x is the initial value of each element of the list.

Default. A null value for i defaults to 0.

Error Conditions. If size of the list is greater than $2^{15}-1$, Error 205 occurs.

Examples

expression	*value*
init := list(5 0)	[0 0 0 0 0]
octave := list(8)	[• • • • • • • •]
count := list(0)	[ ]
*init	5
*octave	8
*count	0

8.1.2 Positional Access to Lists

An element of a list may be accessed by specifying the position of the element in a referencing expression of the form

list left-bracket expr right-bracket

where the value of *expr* is the position of the element in *list* Element positions are also called subscripts Assignment may be made to an element of a list to change its value

Failure Condition A referencing expression fails if the subscript does not reference an element between the 1 and the size of the list, inclusive In this case the subscript is said to be *out of range*

Note Negative subscripts can be used to reference elements relative to the right end of the list For example, −1 references the last element of a list

Examples

expression	*value*
seq[3] = 1	1
seq[5] = seq[3] * 5	5
seq[0]	*none*
seq[−1]	8
seq[−4]	5
unit[1]	*none*

8.1.3 Stack Access to Lists

The functions push(a,x) and pop(a) provide stack access to lists push(a,x) prepends x to the left end of the list a and returns a as its value pop(a) removes the left-most element from the list a and returns this element as its value a[1] is the top of the stack

Failure Condition pop(a) fails if a is empty, that is, if its size is zero

Examples

expression	*value*
laundry = [ ]	[ ]
*laundry	0
push(laundry,"shirts")	[shirts]
push(laundry,"pants")	[pants,shirts]
*laundry	2
pop(laundry)	pants
pop(laundry)	shirts
*laundry	0
pop(laundry)	*none*

8.1.4 Queue Access to Lists

The functions put(a,x) and get(a) provide queue access to lists put(a,x) appends x to the right end of the list a and returns a as its value get(a) removes the left-most element from the list a and returns this element as its value a[1] is the head of the queue For completeness, pull(a) removes the right-most element from the list a and returns this element as its value

Failure Conditions get(a) and pull(a) fail if a is empty

Note pop(a) and get(a) are synonymous

Examples

expression	*value*
laundry := [ ]	[ ]
put(laundry,"shirts")	[shirts]
put(laundry,"pants")	[shirts,pants]
get(laundry)	shirts
get(laundry)	pants
get(laundry)	*none*

8.1.5 Operations on Lists

In addition to the operations above, there are operations for concatenating and sectioning lists

The operation a1 ||| a2 produces the result of concatenating the lists a1 and a2

Note The list produced by a1 ||| a2 is physically distinct from the lists a1 and a2

Examples

expression	*value*
[1,2] ||| [3,4]	[1,2,3,4]
[ ] ||| ["a"]	[a]

Range specifications are used to produce lists that are sections of other lists (see Section 6.1.4)

Notes A list produced by list sectioning is physically distinct from the list to which the range specification is applied. List sections are *not* sublists.

Warning a[i] is the ith element of the list a, it is *not* a list section

Examples

expression	*value*
a := [1,2,3,4]	[1,2,3,4]
a[1:2]	[1]
a[3:0]	[3,4]
a[2+:2]	[2,3]
a[0-:2]	[3,4]

8.2 Tables

A table is an aggregate of elements that resembles a list. A table, however, can be referenced (subscripted) by an object of any type. The elements of a table are not ordered by position. Thus a table can be thought of as an associative list.

8.2.1 Creation of Tables

Tables are created during program execution by the function table(x). When a table is created, it is empty and has no elements. Elements may be added at will and tables grow automatically. Non-existent elements are accessed as if they had the value x.

8.2.2 Accessing Table Elements

An element of a table is accessed by specifying a referencing value in an expression of the form

table left-bracket expr right-bracket

where the value of *expr* references *table*. The referencing value may be of any type. For example t["n"] references the table t with the string n.

Note No type conversion is performed on the value used to reference the table. For example t[1] and t["1"] reference different elements. See also Section 10.3.

A value may be assigned to a table element in a manner similar to that for lists. For example

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

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

A table grows automatically as assignments are made to referenced elements that are not already in the table Table elements are only created, however, when values are assigned to them

The value of *t is the number of elements in the table t

Examples

expression	*value*
op := table()	*table*
*op	0
op["add"] := "c273"	c273
*op	1
op["sub"]	•
op["sub"] := "c274"	c274
*op	2
ct := table()	*table*
ct["four"] := "four"	four
ct["score"] := "twenty"	twenty
*ct	2

8.3 Records

Records are aggregates of variables that resemble lists, but the elements are accessed by name rather than by position

8.3.1 Declaring Record Types

A record type is declared in the form

record *record-name* ([*field-name* [, *field-name*]])

The record name specifies a new type, which is added to the repertoire of types See Section 10 8 The field names provide names by which the fields of the record may be referenced

Notes A record declaration cannot appear within a procedure declaration or within another record declaration The same field name may be used in more than one record declaration and the positions need not be the same Field names do not conflict with identifier names

An example of a record declaration is

```
record complex(real,imag)
```

which declares *complex* to be a record type with two fields, real and imag

8.3.2 Creating Records

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

type (*exp1* [, *exp1*])

where the type is one declared in a record declaration and the values of the expressions provide the values of the fields of the record in the order corresponding to the field names The values may be of any type For example,

```
z := complex(1 0, 2 5)
```

assigns to z a *complex* record with a value of 1 0 for the real field and a value of 2 5 for the imag field

Default Null-valued arguments in a record creation expression default to •

The value of *z is the number of fields declared for the type of record z

8.3.3 Accessing Records

A record is accessed by field name, using the infix . operator Continuing the example above, the value of z real is 1 0 The infix dot operator binds more tightly than any other infix or prefix operator and associates to the left For example, a b c d and ((a b) c) d are equivalent Assignment can be made to a field reference to change the value of that field of the record

Records can also be accessed by position like lists For example, z[1] is equivalent to z real Negative position specifications can be used to access fields relative to the end of the record For example, z[-1] is equivalent to z imag

Failure Condition z[i] fails if the magnitude of i is greater than the number of fields in z

Examples

expression	*value*
z1 = complex(0 0)	*complex*
z2 = complex(3 14, 2 0)	*complex*
z1 real	0
z1 real + z2 imag	2 0
z1 real = z2 real	3 14
z2[2]	2 0
z2[3]	*none*
z1[-2]	3 14

8.4 Sorting Structures

The function sort(a) produces a copy of the list a with the elements in sorted order

In sorting, strings are sorted in non-decreasing lexical order (see Section 6 2), while integers and real numbers are sorted in non-decreasing numerical order (see Sections 4 1 3 and 4 2 3) The ordering of values of other types is unspecified

In heterogeneous lists 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

- integers
- real numbers
- strings
- csets
- files
- procedures
- lists
- tables
- record types

A table is converted to a sorted list by sort(t,i) If the size of t is j, the result is a list of j elements Each element of this list is itself a list of two elements, the first of which is the reference of a table element and the second of which is the corresponding value If i is 1, these two-element lists are in the sorted order of the references of the table If i is 2, these two-element lists are in the sorted order of the values of the table

Note If t is empty, sort(t i) returns an empty list

Default A null value for i defaults to 1

Error Conditions In sort(x) if x is not a list or a table, Error 115 occurs In sort(t,i), if i is not 1 or 2 Error 205 occurs

Chapter 9

Input and Output

9.1 Files

The values of &input, &output, and &errout are the standard input, standard output, and standard error output files, respectively.

Error Condition: These keywords are not variables. If an attempt is made to assign a value to one of them, Error 111 occurs.

A file must be opened to be written or read. 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.

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

The function open(s1,s2) opens the file with name s1 according to the options specified by s2 and returns that file as its value. 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
c	create and open for writing
p	pipe to/from a command (s1 is given to a shell to execute)

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 terminal that behaves like a file that is both written and read.

Warning: File names are interpreted by UNIX. Strange file names may produce strange results.

Default: A null value for s2 defaults to r.

Notes: If a file is opened for writing but not for reading, create is implied. Create and append have no effect on pipes. Pipes may not be opened for simultaneous reading and writing.

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

Error Condition: If the option specification is invalid, Error 209 occurs.

The function close(f) closes f and returns f as its value. 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).

9.2 Writing Data to Files

The function write(x1,...,xn) writes strings to files. Arguments are processed from left to right. If xi is a string or can be converted to one (see Section 5.4), it is written. If xi is a file, subsequent strings are written to that file until another file argument is encountered. Thus strings can be written to several files by a single call of write. Output is written to &output in the absence of a specified file. 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 the strings were concatenated and written as a single line. A line terminator is added after the last string written on each file. The value returned by write is the last string written.

Note: 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.

writes(x1, ,xn) writes in the manner of write(x1, . ,xn), but no line terminators are appended. Thus several strings can be placed on the same line of a file with successive calls of the writes function. One use of this function is to provide prompting at a terminal in interactive mode, allowing the user to respond on the same (visual) line that the inquiry is written.

Defaults. Null-valued arguments for write and writes default to the empty string. ■ If the last argument is a file, an additional ■ is supplied.

Error Condition. If an attempt is made to write on a file that is not open for writing, Error 213 occurs.

Examples

expression	*value*	*value written*	*file written*
out := open("data.txt","w")	*file*	*none*	*none*
flag := "*"	*	*none*	*none*
sep := ":"	:	*none*	*none*
write()	■	■	&output
write(out)	■	■	data.txt
write(out,flag,"a",sep,"b")	b	*a:b	data.txt
write(flag,"a",sep,"b")	b	*a:b	&output
write(out,"x",sep,"y",sep,"z",flag)	*	x:y:z*	data.txt
write(1,sep,2.0,sep,"2")	2	1:2.0:2	&output

9.3 Reading Data from Files

The function read(f) reads the next line from the file f. Line terminators are not included in the returned string.

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. A null value for f defaults to &input.

Note. The maximum input line length is 257. If an input line is longer than 257 characters, only 257 characters are read. Subsequent characters are read on subsequent reads.

Error Condition. If an attempt is made to read from a file which is not opened for reading, Error 212 occurs.

The function reads(f,i) reads the next i characters from the file f. Line terminators are included in the result. If fewer than i characters remain on the file f, the remaining characters are read and the result is shorter than i.

Failure Condition. reads fails if no characters remain to be read.

Defaults. A null value for f defaults to &input. A null value for i defaults to 1.

Note. There is no limit to the maximum value of i except the amount of memory available to store the string.

Error Conditions. If i is less than 1 or greater than $2^{15}-1$, Error 205 occurs. If an attempt is made to read from a file which is not opened for reading, Error 212 occurs.

Chapter 10

Miscellaneous Operations

10.1 Element Generation

The expression !x generates successive elements of x as required. x may be a string, structure, or file.

For strings, successive characters are generated. Assignment to !s may be performed in the same manner as to s[i].

Examples:

expression	*values in sequence*
every !"abcde"	a, b, c, d, e
every !&lcase[10:15]	j, k, l, m, n

For lists, the order of generation is from the first (left-most) element to the last (right-most) element. For example, if a is a list

```
every write(!a)
```

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

For tables, all elements are generated, but the order of generation is unpredictable. For records, the order of generation is the same as for lists. For all structure types, assignment to !x may be used to change the value of an element.

For files, successive lines of input are generated. For example,

```
every write(!&input)
```

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

10.2 Augmented Assignment Operators

One of the commonest operations is the modification of the value of a variable by performing some computation on its previous value. For example

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

increments the value of i.

To simplify such computations, augmented assignment operators are provided in which the computation and assignment operators are combined in a single operator. For example, the value of i is incremented by

```
i +:= 1
```

Note *expr1 +:= expr2* has the same meaning as *expr1 := expr1 + expr2* except that *expr1* is evaluated only once

There are augmented assignment operators for all infix operations except except the assignment operators themselves. For example

```
s ?:= expr
```

scans s and changes its value to the value of *expr*.

Error Condition If the expression on the left side of an augmented assignment operator is not a variable, Error 111 occurs

10.3 Comparison of Objects

Most comparison operations such as i = j and s1 == s2 are concerned with comparison of values. In these cases implicit type conversion occurs prior to the comparison.

The two operations x === y and x ~=== y are concerned with the equivalence of objects. x === y succeeds if x and y are of the same type and are equivalent. Similarly, x ~=== y succeeds if x and y are of different types or if they are not equivalent. In both cases, the value of the right operand is returned in the case of successful comparison.

The meaning of the term "equivalent" as used here depends on the type. Integers, real numbers, strings and csets are considered to be equivalent if they have the same values, regardless of how they are computed. For procedures, files, lists, tables, and record objects, object comparison fails regardless of value, unless x and y are the *same* object.

Note. The kind of comparison used in x === y is also used to determine whether two table references are the same. See also Section 8.2.2.

Examples

expression	*value*
("abc" || "def") === "abcdef"	abcdef
7 === (6 + 1)	7
7 === "7"	*none*
'amy' === 'may'	a m y
[10,10] === [10 10]	*none*
{x := y := list(10) x === y}	*list*

10.4 Copying Objects

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

```
a1 := list(10)
a2 := a1
```

assign the same list to a1 and a2. Subsequently, a1[3] and a2[3] reference the same element of the same list.

An object may be copied by the function copy(x). For example, if a1 is a list,

```
a2 := copy(a1)
```

assigns a copy of a1 to a2. This copy is the same size as a1 and the values of all the elements are the same, but a1 and a2 are distinct objects. Subsequently, a1[3] and a2[3] reference elements in the corresponding positions of different objects.

Note. Any type of object may be copied. In the case of integers, real numbers, strings, files, procedures, csets, and •, the result is not a physically distinct object, but this difference is undetectable. See Section 10.3.

10.5 Random Element Generation

The operation ?x returns a randomly selected value from x. If x is a positive integer i, ?x produces an integer from a pseudo-random sequence in the range of 1 ≤ ?x ≤ i. ?0 produces a real number r from a pseudo-random sequence in the range 0.0 < r < 1.0.

If x is a string, ?x returns a randomly selected one-character substring of x.

If x is a list, table, or record, ?x returns a randomly selected element of x.

Note. For structures, variables are produced and assignment can be made to them.

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.

Error Condition: If the value of i in ?i is less than 0 or greater than $2^{15}-1$, Error 205 occurs.

10.6 Date and Time

The value of the keyword &date is the current date in the form yyyy/mm/dd. For example, the value of &date for December 1, 1981 is 1981/12/01.

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 &dateline is the date and time of day in a readable format. An example is Friday, December 4, 1981 7:42 am .

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

Note: The value of &time includes only user time, not system time.

Error Condition: &date, &clock, &dateline, and &time are not variables. If an attempt is made to assign a value to one of them, Error 111 occurs.

10.7 The Null Value

The null value, •, is the initial value of all identifiers and is provided as the value for omitted expressions in function and procedure calls, as well as in some control structures. In addition, the value of the keyword &null is •.

The null value is illegal in most computational contexts, although it defaults to commonly used values for the arguments of some functions. See Appendix C.

There are two operations that can be used to test for •:

/*expr* returns *expr* if the value of *expr* is •, but fails otherwise.

expr returns *expr* if the value of *expr* is not •, but fails otherwise.

Note: If *expr* produces a variable, these operations return that variable. For example, /v := 0 assigns 0 to v if the value of v is •.

10.8 Type Determination

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

Examples:

expression	*value*
type(1)	integer
type(2.0)	real
type("")	string
type('armada')	cset
type(trim)	procedure
type(main)	procedure
type()	null

10.9 String Images

The function image(x) produces a string that represents the value of x. For strings and csets, this includes enclosing quotes and escapes as necessary. For structures, their current size is given. Keywords are given in place of their values in several cases.

Examples

expression	*value*
image(1)	1
image(2.0)	2.0
image("abc")	"abc"
image("")	""
image('drama')	'admr'
image(&lcase)	&lcase
image()	&null
image(&input)	&input
image(open("data","w"))	file(data)
image([1,0,11])	list(3)
image(list(10))	list(10)
image(complex(3.1,1.0))	record complex(2)
image(trim)	function trim
image(main)	procedure main
image(complex)	record constructor complex

Note Note that image(x) can be used to distinguish between functions, procedures, and record constructors.

10.10 Calling a Shell

The function system(s) calls a shell to execute the string s. For example, system("ls") lists the current directory. The value returned by system(s) is the exit status returned by the shell.

Error Condition If the size of s is greater than 256, Error 210 occurs.

10.11 System Information

The value of the keyword &host is the host location, operating system, and computer on which Icon is running. An example is University of Arizona, UNIX Version 7, PDP-11/70.

The value of the keyword &version is the name and version number of the Icon implementation. An example is Icon Version 5.0 interpreter, December 1981.

Error Condition &host and &version are not variables. If an attempt is made to assign a value to one of them, Error 111 occurs.

Chapter 11

Procedures

11.1 Procedure Declarations

A procedure declaration has the form

```
procedure identifier ( [ identifier [ , identifier ]    ] ) ;
   [ local-declaration ; ]
   [ initial-clause ; ]
   [ procedure-body ; ]
end
```

Note The semicolons in a procedure declaration may be omitted if the components are placed on separate lines. See also Section 12.2.

The identifier following **procedure** gives the name of the procedure. A local declaration has the form

local-specification identifier [, *identifier*]

A local specification may be **local**, **dynamic**, or **static**.

Note **local** and **dynamic** are equivalent.

Examples

```
local x, y
dynamic count
static state, basis
```

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.

Note Identifiers in the argument list are dynamic.

The initial clause has the form

initial *expr*

The expression in the initial clause is evaluated once when the procedure is called the first time. The initial clause is useful for assigning values to static identifiers.

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

Two examples of procedure declarations follow.

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

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

11.2 Scope of Identifiers

As indicated in the preceding section, identifiers declared in a procedure are accessible only to that procedure. If an identifier in a procedure is not declared, its scope is determined by **global** declarations that apply to the entire program.

global *identifier* [, *identifier*]

specifies that the listed identifiers are to be interpreted as global in those procedures in which they are not explicitly declared to be local. The values of such variables are accessible to all such procedures.

Notes: A local declaration for an identifier in a procedure overrides a global declaration for that identifier. Global declarations cannot occur inside other declarations but they otherwise may occur anywhere in the program. Record names have global scope, but this scope can be overridden by local declarations. Field names are not identifiers; they apply to the entire program and are not affected by scope declarations.

The scope of an identifier for which there is neither a local nor a global declaration is local.

11.3 Procedure Activation

11.3.1 Procedure Invocation

Procedures are invoked in the same form that functions are called.

expr ([*expr* [, *expr*]])

where the expression before the parenthesized list has a procedure value. This expression usually is an identifier. For example, the procedure **max** given in the example above might be used as follows:

```
m := max(*x,*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.

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, • is provided for the remaining arguments.

11.3.2 Return from Procedures

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

return [*expr*]
fail
suspend [*expr*]

Defaults: An omitted *expr* in a return expression defaults to •. If control flows off the end of a procedure body without an explicit return, the procedure call returns no result (that is, it fails).

Warning: Failure to provide an explicit return from a procedure body may lead to unexpected and erroneous results.

The expression **return** *expr* terminates the call of a procedure and returns the outcome 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(*x,*y)
```

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

The expression **fail** terminates the call of a procedure without returning a result, causing the calling expression to fail. Consider the following procedure:

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

This procedure compares the types of x and y, returning • if they are the same and failing otherwise On the other hand

```
return type(x) == type(y)
```

also fails if the types are not the same, but returns the type instead of • if the types are the same

The expression **suspend** *expr* is similar to **return** *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 the context in which the procedure call occurs requires an alternative result Thus suspended procedures are generators Consider the following procedure

```
procedure timer(t)
   while &time < t do suspend
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

Like **every** **suspend** 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**

Note The **suspend** expression itself fails once all alternatives of *expr* have been produced

If the expression in **return** or **suspend** is a global identifier or a computed variable (such as a list element), the variable is not dereferenced Local identifiers are dereferenced, however, and only their 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,i,j)
   if x[i] > x[j] then return x[i]
   else return x[j]
end
```

An assignment to a call of this procedure such as

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

changes the value of the maximum of the elements k and m in roster

11.3.3 Procedure Level

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

Error Conditions 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 304 occurs **&level** is not a variable If an attempt is made to assign a value to it, Error 111 occurs

11.3.4 Tracing Procedure Activity

Tracing of procedure invocation is controlled by the keyword **&trace** If the value of **&trace** is nonzero a diagnostic message is written to **&errout** 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

Notes. Tracing stops automatically when &trace is decremented to 0. If a negative value is assigned to &trace, tracing continues indefinitely. If the value assigned to &trace is less than -2^{15} or greater than $2^{15}-1$, the actual value assigned is -1.

In the case of a procedure call, the trace message includes the name of the procedure and string images of the values of its arguments. The message is indented with a number of vertical bars 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. All trace messages include the name of the file containing the procedure that is traced and the line number in that file from which the call or return is made.

An example is given by the following program, which is contained in the file acker.icn.

```
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

```
acker.icn:10    | acker(1,3)
acker.icn:5     | | acker(1,2)
acker.icn:5     | | | acker(1,1)
acker.icn:5     | | | | acker(1,0)
acker.icn:4     | | | | | acker(0,1)
acker.icn:3     | | | | | acker returned 2
acker.icn:4     | | | | acker returned 2
acker.icn:5     | | | | acker(0,2)
acker.icn:3     | | | | acker returned 3
acker.icn:5     | | | acker returned 3
acker.icn:5     | | | acker(0,3)
acker.icn:3     | | | acker returned 4
acker.icn:5     | | acker returned 4
acker.icn:5     | | acker(0,4)
acker.icn:3     | | acker returned 5
acker.icn:5     | acker returned 5
acker.icn:11    main failed
```

Note that the procedure main, which has no explicit return, produces no result (that is, it fails).

In trace output, values are imaged in a manner similar to that produced by image(x) (see Section 10.8). In order to prevent trace output from being unwieldy, literal strings and csets are truncated to 16 characters and followed by ellipses (...) to indicate the truncation. For lists and records, values are shown for up to six elements. If the size of a list or record is greater than six, the first three and last three elements are shown, with ellipses indicating the omitted elements. Various additional information is shown, such as where variables are returned and the ranges for substrings.

11.4 Listing Identifier Values

The function display(i,f) prints a list of all identifiers and their values in the i levels of procedure invocation starting at the current procedure invocation The output is written to f

Notes display(&level f) displays the identifiers in all procedure invocations leading to the current invocation display(0,f) displays only global identifiers display(i,f) returns • as its value

Defaults A null value for i defaults to &level A null value for f defaults to &errout

Error Condition If the value of i is less than 0, Error 205 occurs

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

```
global hexd

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

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

The output of display(&level) is

```
hex local identifiers
   x = "61"
main local identifiers
   label = "hex(61)="
global identifiers
   main = procedure main
   hexd = "0123456789ABCDEF"
   hex = procedure hex
   write = function write
   display = function display
   find = function find
```

Global identifiers are listed at the end of every display output, regardless of whether or not the global identifiers are referenced by the displayed procedures

11.5 Procedure Names and Values

A procedure declaration establishes an object of type procedure as the initial value of the global 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 are procedures, such as

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

then

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

computes the maximum of i and j

The names of functions are global identifiers with predefined values The declaration of a procedure or record with the same name as a function overrides the predefined value A local declaration for a function

name has the same effect within the procedure in which the declaration occurs

11.6 External Procedures

Procedures written in C can be included in an Icon program by the declaration

external *identifier* [, *identifier*]

where *identifier* is the name of a C procedure. External procedures have the same status as Icon functions. See Reference 12 for coding conventions that must be used in writing external procedures.

Chapter 12

Program Preparation

12.1 Program Structure

A program is a sequence of declarations. The declarations may appear in any order. The executable components of a program are contained in procedure declarations. Every program must contain a procedure named main.

A program may be divided into a number of files, but every declaration must be completely contained in a single file. When a multi-file program is processed, the scope of identifiers is the same as if the program had been contained in a single file.

Warning. A global declaration in one file of a program may affect the interpretation of an undeclared identifier in another file.

Note. Record and procedure declarations implicitly declare their record and procedure names respectively to be global.

12.2 Layout of Program Text

Since a file is a sequence of lines, it is usually convenient and natural to parallel the logical structure of a sequence of expressions by the physical structure of a sequence of lines in the file.

Semicolons are used in a number of places to separate expressions. See Appendix A. If a semicolon falls at the end of a line, it may be omitted, provided that the syntactic token at the end of the line can legitimately end an expression and the token at the beginning of the next line can legitimately begin an expression. Thus most semicolons can be omitted at the ends of lines, and long expressions can be written on several lines without difficulty.

Note. If a semicolon can be legitimately inserted in the place of a newline character in program text, this is done automatically by the Icon translator.

For example

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

can also be written as

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

Because of the way that the translator interprets ends of lines, if an infix operation is split into two lines, the operator should be placed at the end of the first line, not at the beginning of the second. For example

```
s1 ||
s2
```

is the concatenation of the values of two identifiers, while

```
s1
|| s2
```

is two expressions, the first of which is a lone identifier and the second of which is two repeated alternations of a second identifier!

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

```
return expr
```

and

```
return
expr
```

are quite different

Identifiers may be arbitrarily long, but must be contained on one line. A quoted 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.

Note. The total length of a string literal is limited only by the memory available to the translator. There is no practical limit.

12.3 Program Character Set

Icon uses the ASCII character set [11]. In program text, tabs and blanks are syntactically equivalent. All other characters are distinct.

Note. In literal strings, blanks and tabs are distinct.

12.4 Significance of blanks

Blanks (and tabs) in program text, except in string literals, serve to separate tokens that otherwise would appear to be a single token. Blanks are otherwise optional between tokens and may be used for indentation and to produced desired visual effects in program text. Blanks are necessary to separate reserved words, identifiers, and where an infix operator that is followed by a prefix operator would be ambiguous. For example

```
x--y
```

is interpreted as the character set difference of x and y, while

```
x- -y
```

is interpreted as x minus the negative of y.

12.5 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 illustrates the use of comments.

```
# These procedures print all the intersections of two words
# cross uses nested every constructs to find all intersections and
# calls xprint to print each intersection

procedure cross(word1, word2)
   local j, k
   every j := upto(word2, word1) do          # location in word1 of
                                             # every character in word2
      every k := upto(word1[j], word2) do    # and for each, all
                                             # positions in word2
         xprint(word1, word2, j, k)          # print the result
end

procedure xprint(word1, word2, j, k)
   every write(right(word2[ 1 to k - 1 ], j))   # up to position in word1
   write(word1)                                 # then word1
   every write(right(word2[ k + 1 to *word2 ], j))   # then rest of word2
end
```

Chapter 13

Programming Considerations

13.1 Efficiency Considerations

Many of the considerations in writing efficient Icon programs are the same as for other languages: use of good algorithms, good program structure, appropriate data representations, and so on. There are, however, idiosyncrasies of the Icon language and its implementation that warrant specific attention:

1. Any operation that causes the allocation of a significant amount of storage may adversely affect running speed, since that storage must eventually be reclaimed by garbage collection, a relatively expensive process. While a detailed understanding of storage allocation and garbage collection requires extensive knowledge of the implementation of Icon, common sense provides a good guide to programming practices. Some specific aspects of storage allocation are mentioned below.

2. Long strings are expensive to manipulate. Operations that construct strings require storage allocation and the movement of data. Appending to the last string constructed is a comparatively inexpensive process, however.

3. Creation of a substring does not require a significant amount of storage and involves no movement of data. Assignment to a substring, however, is a form of string creation.

4. Several strings can be appended in output without concatenation by using write and writes. This technique frequently can be used to avoid considerable amounts of storage allocation. Note that multi-line output can be produced in a single output expression by using "\n" to generate newlines.

5. Icon stores integers in the range of -2^{15} to $2^{15}-1$ in one word. One-word integers do not require the allocation of storage. For integers beyond this range, two words are used. Two-word integers do require the allocation of storage.

6. Icon provides automatic type conversion (coercion) where possible. Such type conversions, although not directly evident, may be the cause of significant inefficiencies. The worst potential problems are in cset-to-string and string-to-cset conversion. For example, evaluation of upto("aeiou") causes the string aeiou to be converted to a cset every time the expression is evaluated. If such an expression occurs in a frequently executed inner loop, overall program performance may be significantly affected. It is good programming practice to use cset literals or to perform an explicit out-of-line conversion in such cases.

7. Augmented assignment operations, such as i +:= 1, should be used wherever possible to avoid two evaluations of the variable to which the assignment is made. This is particularly important in the case of table references (for example, t["n"] +:= 1), since table references are comparatively slow.

8. Case selector expressions are evaluated in the order in which they appear (except for default). Consequently, selector expressions should be ordered according to likelihood of selection.

9. Compound comparisons should be ordered so that unnecessary comparisons are avoided if the final outcome is failure. For example

```
0 = f(x) = g(x)
```

is generally more efficient than

```
f(x) = g(x) = 0
```

since f(x) and g(x) may produce the same, but nonzero, value This consideration is particularly important when expressions in the comparison may have many alternative results

13.2 Programming Pitfalls

Since Icon has several unusual features, the novice Icon programmer is likely to encounter a number of problems that would not come up in other programming languages Some of the problems that may be encountered are described below

1 Generators are reactivated for successive alternatives in a last-in first-out manner As a result, all possible alternative results are produced, if necessary, in the goal-directed mode of evaluation used by Icon However, the order of evaluation that results from last-in, first-out reactivation of generators is different from that in conventional left-to-right, precedence-determined evaluation of expressions In particular, if a generator is reactivated for an alternative result, only those components of the expression that follow the reactivated generator are re-evaluated If generators are used in complicated combinations, unexpected results may occur for these reasons In particular, it is bad programming practice to use generators to produce side effects in an **every** clause

2 The referencing expression **x[y]** is polymorphous, allowing **x** to be a string, list, table, or record object If **x** is not of the type that is expected, unusual results may occur In particular, it is a common programming practice for **x** to be a list and for an expression of the form **x := x[i]** to be used to link through a structure If **x[i]** is a string instead of a list (perhaps as a result of an error in building the structure), an endless loop may result

3 Assignment does not copy structures Thus, if **a1** is a list, **a2 = a1** assigns the *same* list to **a2** Thus assignment to an element of **a1** changes that element of **a2** Similarly, the effect of

```
a = list(3,list(5))
```

is to assign the *same* list of five elements to each of the three elements of **a**

4 Exiting string scanning whether by **next**, **break**, or a procedure return, does not restore the previous values of **&subject** and **&pos** Unless this effect is specifically desired or known to be safe, it is not good practice to exit from string scanning

5 Since return from a procedure by flowing off the end of the procedure body causes the call of the procedure to fail unexpected results may occur if the call is used in a context where its outcome is significant Such failure may cause an enclosing expression to fail If the call is in a goal-directed context, the function may be called again for other values of its arguments

6 Since dereferencing is not performed until all arguments of a function or operation are evaluated, unexpected results may occur if side effects change the values of variables during argument evaluation For example

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

writes **aa** regardless of the value of **s** prior to the evaluation of the **write** function The explicit dereferencing operator **.** may be used to avoid this problem

7 Since the outcome of loop control structures is failure, their use in contexts where this failure is significant may produce unexpected results For example, if *expr2* in *expr1* **?** *expr2* is such a control structure the entire scanning expression fails Similarly, if *expr1* **then** *expr2* fails if *expr1* fails

8 In *expr1* **?** *expr2* neither *expr1* nor *expr2* is limited in the number of results it may produce in a goal directed context In particular if *expr2* fails, backtracking to *expr1* occurs

9. The functions move(i) and tab(i) restore the value of &pos if they are activated to produce an alternative result. If this effect is not anticipated, the consequences may be mysterious. For example

```
suspend move(1)
```

produces only one result, but if an alternative is sought (by goal-directed evaluation at the cite where the procedure containing this suspend is called), &pos is restored.

10. Since • is illegal in most computational contexts, failure to assign an appropriate value to a variable before it is used usually results in a run-time error.

11. The names of functions are global identifiers with predefined values. If such a name is declared to be local in a procedure, it may be used as an identifier like any other name, but the corresponding function is inaccessible within that procedure. If such a declaration is made unintentionally, the results may be mysterious.

12. In splitting long program lines, binary operators should be placed at the end of line, not the beginning. Otherwise the translator may interpret the lines as syntactically correct, but differently from the way intended by the programmer.

13. SNOBOL4 programmers are prone to omit the || operator that is required for concatenation in Icon. The result is usually a syntax error. A more subtle error is the use of = in place of := for assignment. This error may produce undetected program malfunction or a run-time type error.

Chapter 14

Running Icon Programs

There are four phases in processing an Icon program: translation, linking, loading, and execution.

14.1 Translation

An Icon program is first translated into an intermediate form. The translator may 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 multiply declared identifiers. See Appendix D for a list of translator 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 a syntactic error, the translation process is continued, but the program is not executed. There are also overflow conditions that cause termination of translation at the point of overflow. See Appendix D.

14.2 Linking

Once an Icon program has been translated into its intermediate form, there is a linking phase in which the scope of identifiers is resolved and in which a form suitable for execution is produced.

In the C implementation, there are two options: interpretation and compilation. The linker for the interpreter produces a compact representation of the program that is executed interpretively. The linker for the compiler produces executable machine language. The translation and linking processes for the interpreter are fast and the program sets into execution quickly. Compilation is considerably slower, but the code it produces executes somewhat faster. One advantage of the compiler is that it allows separately translated program segments to be linked together and external procedures to be included. In order to produce executable code, the compiler has additional assembly and loading phases. Loading ("link editingd") is done by the UNIX program *ld* [10]. At this time external procedures are added to the Icon run-time system and linked program.

The error message **text overflow** from *ld* indicates that there is not enough memory available to run the Icon program.

14.3 Program Execution

Program execution is initiated by invoking the procedure **main**.

If there are any arguments on the UNIX command line used to initiate program execution, **main** is invoked with one argument, which consists of a list of strings. Each string corresponds to one argument on the command line (not including the "zeroth" argument).

Note: If there is no argument on the command line, **main** is invoked with an empty list.

14.4 Program Termination

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

Note: The exit status on return from **main** is 0.

Program termination may also be caused by **stop(x1, ..., xn)**. The function **stop** writes in the fashion of **write** (see Section 9.2) and then causes termination.

Notes 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 during program execution. stop produces an exit status of 1

Default If the first argument to stop is not a file, output is written to &errout until a file argument is encountered

The function exit(i) terminates program execution with an exit status of i.

14.5 Error Termination

Errors that occur during program execution may result from logical mistakes, invalid data, and so forth. If such an error occurs, an error number and an explanatory message are printed. In some cases, the offending value is shown. See Appendix D for a list of run-time error messages. A run-time error terminates program execution with an exit status of 2.

Chapter 15

Sample Programs

This chapter contains a number of sample programs. These programs illustrate various aspects of programming in Icon. No claim is made that the programming techniques or the algorithms used here are the best, but they are all running programs and they were written by programmers who have used Icon for some time.

The programs are preceded by problem statements and discussions of the methods used for the solutions. Discussions follow the programs. Icon idioms and points of special interest are noted. Exercises include suggested extensions, improvements, and related problems.

The programs themselves have been stripped of most comments for better typographic presentation. In most cases, error checking and embellishments have been omitted also. These amenities can be provided by the interested reader.

All the programs in this chapter are included in the Icon distribution system for UNIX.

15.1 Roman Numerals

Description: This problem is a simple one: write a program to convert Arabic numerals to corresponding Roman numerals.

Solution: The method of solution is due to Gimpel [13]. Each digit of the Arabic number is mapped into its Roman equivalent. The multiplication by 10 represented by successive positions in the Arabic number is reflected in the corresponding Roman numeral by shifting to the next "octave" using character replacement. The occurrence of an asterisk in the result indicates a number that is too large to be represented by a Roman numeral.

```
#
#          R O M A N   N U M E R A L S
#

#  This main procedure takes Arabic numerals from standard input and writes
#  the corresponding Roman numerals to standard outout.

procedure main()
   local n
   while n := read() do
      write(roman(n) | "cannot convert")
end

procedure roman(n)
   local arabic, result
   static equiv
   initial equiv := ["","I","II","III","IV","V","VI","VII","VIII","IX"]
   integer(n) > 0 | fail
   result := ""
   every arabic := !n do
      result := map(result,"IVXLCDM","XLCDM**") || equiv[arabic+1]
   if find("*",result) then fail else return result
end
```

Exercises:

1. Rewrite the **every** loop to eliminate the local identifier **arabic**
2. Modify **equiv** so that the addition of 1 is not necessary when it is referenced.
3. Consider alternative data representations for **equiv**, including strings and tables.
4. Write a procedure to convert Roman numerals to Arabic numerals.

15.2 Meandering Strings

Description: A string over an alphabet of k characters is said to be an n-meander if it contains every possible substring of length n from the alphabet [14]. For example, 0001111011001010000 is a 4-meander for the alphabet 01.

The problem here is to write a procedure to compute meandering strings of minimal length (the example given above is minimal).

Solution: In Reference 14, it is shown that the length of the minimal meandering string is k^n+n-1 and an algorithm is given to generate such a string. The algorithm is basically an enumerative one, systematically constructing substrings, but discarding ones that already occur in the result.

```
#
#          M E A N D E R I N G   S T R I N G S
#

#  This main procedure accepts specifications for meandering strings
#  from standard input with the alphabet separated from the length by
#  a colon.

procedure main()
   local line, alpha, n
   while line := read() do {
      line ? if alpha := tab(upto(':')) then {
         move(1)
         if n := integer(tab(0)) then write(meander(alpha,n))
         else write("erroneous input")
         }
         else write("erroneous input")
      }
end

procedure meander(alpha,n)
   local result, t, i, c, k
   i := k := *alpha
   t := n-1
   result := repl(alpha[1],t)
   while c := alpha[i] do {
      if find(result[-t:0] || c,result)
      then i -:= 1
      else {result ||:= c; i := k}
      }
   return result
end
```

Exercises:

1. Try to improve the algorithm used in the solution above.

2 Apply the concept of meandering strings to produce space-efficient techniques for telegraphic codes

15.3 Word Intersections

Description Given two strings, display their intersections in common characters

Solution The approach is to consider one string as a set of characters and look for occurrences of these characters in the other string

```
#
#          W O R D   I N T E R S E C T I O N S
#

#  This main procedure accepts word pairs from standard input, with
#  the words separated by semicolons.

procedure main()
   local line, j
   while line := read() do {
      write()
      j := upto(':',line)
      cross(line[1:j],line[j+1:0])
      }
end

procedure cross(s1,s2)
   local j, k
   every j := upto(s2,s1) do
      every k := upto(s1[j],s2) do
         xprint(s1,s2,j,k)
end

procedure xprint(s1,s2,j,k)
   write()
   every write(right(s2[1 to k-1],j))
   write(s1)
   every write(right(s2[k+1 to *s2],j))
end
```

Comments The procedure cross(s1,s2) provides a good illustration of generators and particularly how nested generators can be used to formulate a search over many alternatives The procedure xprint(s1,s2,j,k) prints s1 horizontally and s2 vertically, crossing at the point of intersection For example, the output of cross("fish","school") is

```
f i s h
    c
    h
    o
    o
    l

      s
      c
f i s h
      o
      o
      l
```

Exercises.

1. Extend the solution to handle the mutual intersections of several words.
2. Extend the solution to the generation of Kriss-Kross puzzles [15]

15.4 Word Counting

Description: One of the simplest illustrations of the utility of string scanning, as opposed to more primitive string analysis methods, is counting the words contained in a file of text. For the purposes of this problem, a "word" is defined to be a sequence of letters. The output is a listing of words in alphabetical order, together with a count of the number of times each word occurs in the file.

Solution: String scanning tabs up to a letter. The subsequent sequence of letters references a table and the count is incremented. When processing of the file is complete, the table is sorted and printed, using a column width that is supplied as an argument to the procedure. The text to be processed comes from standard input and the results are written to standard output.

```
#
#          W O R D   C O U N T I N G
#

#  This main procedure processes standard input and writes the results
#  with the words in a column 20 characters wide.

procedure main()
   wordcount(20)
end

procedure wordcount(n)
   local t, line, x, y
   static letters
   initial letters := &lcase ++ &ucase
   t := table(0)
   while line := read() do
      line ? while tab(upto(letters)) do
         t[tab(many(letters))] +:= 1
   x := sort(t)
   every y := !x do write(left(y[1],n),y[2])
end
```

Comments. Note the use of augmented assignment to update the count without having to reference the table twice.

Exercises:

1. Modify the solution so that a suitable column width is computed by the procedure wordcount
2. Revise the solution so that the output is ordered by decreasing count.
3. Revise the solution so that the output is broken down into sections of words having the same count and with the words listed alphabetically in each section

15.5 Binary Trees

Description: Write a program to construct and traverse binary trees

Solution The nodes in a binary tree can be represented by records, in which one field is devoted to the contents of the node and two other fields point to the left and right subtrees. For input output purposes, trees are represented by strings in which parentheses and commas specify the skeleton of the tree and the contents of the nodes are given between punctuation characters. For example, a(b,c) represents a tree with a root node containing a and two leaves containing b and c, respectively.

```
#
#          B I N A R Y   T R E E S
#

#  This main program accepts string representations of binary trees from
#  standard input  It performs a tree walk and lists the leaves of
#  each tree

record node(data,ltree,rtree)

procedure main()
   local line, tree
   while line := read() do {
      tree := tform(line)
      write("tree walk")
      every write(walk(tree))
      write("leaves")
      every write(leaves(tree))
      }
end

procedure tform(s)
   local value,left,right
   if /s then return
   s ? if value := tab(upto('(')) then {
      move(1)
      left := tab(bal(','))
      move(1)
      right  = tab(bal(')'))
      return node(value,tform(left),tform(right))
      }
      else return node(s)
end

procedure walk(t)
   suspend walk(\t ltree | \t rtree)
   return t data
end

procedure leaves(t)
   if not(\t ltree | \t rtree) then return t data
   suspend leaves(\t ltree | \t rtree)
end
```

Comments The procedure **tform** constructs the binary tree from a string representation of the type described above The procedures **walk** and **leaves** walk the tree and generate the leaves, respectively Note that these procedures are generators allowing successive nodes to be obtained as desired

Exercises

1 Modify the procedure **tform** to allow trailing commas to be omitted to indicate the absence of a right subtree

2 Modify the procedure **walk** to walk the tree in various different orders

3 Add error checking to the procedure **tform** to detect syntactically incorrect input

4 Write a procedure to convert a binary tree into its string representation.

15.6 Eight Queens

Description: The classic example used to illustrate backtracking is the eight-queens problem [18,19], which is to determine the number of ways that eight queens can be placed on a chess board such that none can attack another.

Solution: The solution involves trial placements of the eight queens with backtracking from attacking positions.

```
#
#          E I G H T   Q U E E N S
#

procedure main()
   every write(q(1),q(2),q(3),q(4),q(5),q(6),q(7),q(8))
end

procedure q(c)
   suspend place(1 to 8,c)
end

procedure place(r,c)
   static up, down, rows, upoff, downoff
   initial {
      up := list(15,0)
      down := list(15,0)
      rows := list(8,0)
      upoff := 8
      downoff := -1
      }
   if rows[r] = up[upoff+r-c] = down[downoff+r+c] = 0 then
      suspend rows[r] <- up[upoff+r-c] <- down[downoff+r+c] <- r
end


procedure q(c)
   suspend place(1 to 8,c)
end

procedure place(r,c)
   static up, down, rows, upoff, downoff
   initial {
      up := list(15,0)
      down := list(15,0)
      rows := list(8,0)
      upoff := 8
      downoff := -1
      }
   if rows[r] = up[upoff+r-c] = down[downoff+r+c] = 0 then
      suspend rows[r] <- up[upoff+r-c] <- down[downoff+r+c] <- r
end
```

Comments: The three lists keep track of the free rows, the upward-facing diagonals, and the downward-facing diagonals. Free squares are indicated by zero values, while occupied squares are indicated by the value one. Note that goal-directed evaluation forces the function write to be called for all combinations of

arguments that have values (for which q(i) returns a value).

Exercises:

1 Write an analogous procedure for four rooks.

2 Write a procedure to display the solutions in the format of a chess board.

15.7 Infix-to-Prefix Conversion

Description: Write a program to convert arithmetic expressions from infix form to fully parenthesized prefix form. The desired conversions are illustrated by the following examples:

x	x
x+1	+(x,1)
((x+1))	+(x,1)
x-y-z	-(-(x,y),z)
3*delta+1	+(*(3,delta),1)
2^2^n	^(2,^(2,n))
(x^n)/(z+1)	/(^(x,n),+(z,1))

Solution: Since the infix expressions may not be fully parenthesized, the precedence and associativity of the infix operators must be considered. In addition, the infix expressions may contain superfluous parentheses that must be removed. Separate procedures are provided to remove such superfluous parentheses and for handling left- and right-associative operators according to their conventional precedences. Once an expression has been decomposed into its operators and operands, the corresponding prefix expression is easily obtained

```
#
#          I N F I X - T O - P R E F I X   C O N V E R S I O N
#

#  This main procedure accepts infix expressions from standard input and
#  writes the corresponding prefix expressions to standard output.

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

procedure prefix(s)
   s := strip(s)
   return lassoc(s,'+-' | '*/') | rassoc(s,'^') | s
end

procedure strip(s)
   while s ? (="(" & s <- tab(bal(')')) & pos(-1))
   return s
end

procedure lassoc(s,c)
   local j
   s ? every j := bal(c)
   return form(s,\j)
end
```

```
procedure rassoc(s,c)
   local j
   return form(s,s ? bal(c))
end

procedure form(s,k)
   local a1, a2, op
   s ? {
      a1 := tab(k)
      op := move(1)
      a2 := tab(0)
      }
   return op || "(" || prefix(a1) || "," || prefix(a2) || ")"
end
```

Comments This solution illustrates a number of facets of string scanning and the use of the function **bal** in particular Note the use of conjunction in **strip** to assure that the balanced string ends at a terminal parenthesis

Exercises

1 Modify the procedure **prefix** to avoid calling **lassoc** and **rassoc** in case **s** does not contain any operators

2 Write a procedure to convert from prefix form to infix form

3 Extend the solution given above to handle prefix operators and functional forms

4 Write a program to perform symbolic differentiation

5 Write a program to perform general symbolic evaluation Provide for simplification of the results

15.8 Recognition of Context-Free Languages

Description Given a context-free grammar, write a program to recognize sentences from the corresponding language

Solution In SNOBOL4 there is an isomorphism between the productions of a context-free grammar and corresponding recognition patterns [20] Provided there is no left recursion, there is a similar isomorphism in Icon, in which recognition procedures take the place of patterns This isomorphism is illustrated by the following simple grammar

```
<s> ::= a <s> | <t> b | c
<t> ::= d <s> d | e | f
```

A program containing recognition procedures **s** and **t** corresponding to <s> and <t> follows

```
#
#          C F L   R E C O G N I T I O N
#

#  This main procedure takes strings from standard input and determines
#  whether or not they are sentences in the language defined by <s>

procedure main()
   local line
   while line := read() do
      if recogn(s,line) then write("accepted") else write("rejected")
end
```

```
procedure recogn(goal,text)
   return text ? (goal() & pos(0))
end

#  <s> ::= a <s> | <t> b | c

procedure s()
   suspend (="a" || s()) | (t() || ="b") | ="c"
end

#  <t> :.= d <s> d | e | f

procedure t()
   suspend (="d" || s() || ="d") | ="e" | ="f"
end
```

Comments Terminal symbols are matched by expressions of the form =x, while nonterminal symbols are matched by calls on the corresponding recognition procedures. For each successful match, a recognition procedure suspends with the value matched.

The procedure **recogn** succeeds or fails, depending on whether or not **text** is a sentence in the **goal** grammar. Note that the goal procedure is an argument of **recogn**. This demonstrates the usefulness of procedures being data objects.

The use of conjunction and a test for a position at the end of **&subject** are necessary to prevent spurious recognition of an initial substring.

Exercises

1. Note that the recognition procedures return the substring that they match. Run the program with tracing and various input, observing how the recognition process proceeds.

2. Write a program to accept a grammar as input and generate corresponding recognition procedures.

3. Procedures of the type used here are not limited to recognition. Adapt them to the generation of parse trees.

15.9 Random Sentence Generation

Description Write a program to accept a context-free grammar as input and generate randomly selected sentences from the corresponding language.

Solution The solution here is patterned after the one given in Reference 21, which should be consulted for a more detailed description.

Grammatical specifications are read in and analyzed. A list of alternatives is created for each definition. Each alternative, in turn, is represented by a list of subsequents (terminal and nonterminal symbols). The name of a nonterminal is associated with its structure through a table. Terminals are represented by strings, while nonterminals are represented by records.

Generation specifications are represented by a nonterminal followed by a count. For example, <s>10 specifies the generation of 10 sentences from the language defined by <s>.

The generation process starts with a generation list consisting of the desired nonterminal. Elements are removed from the left end of this list. If an element is a nonterminal, the subsequent list for one of its randomly selected alternatives is prepended to the generation list. If an element is a terminal, it is appended to the evolving result.

```
#
#          R A N D O M   S E N T E N C E   G E N E R A T I O N
#

global def

record nonterm(ntname)

procedure main()
   local line
   def := table()
   while line := read()do
      enter(line) | generate(line) | write("*** syntax error")
end

procedure enter(s)
   local name
   return s ?
      if ="<" then {
         name := tab(find(">::=")) | fail
         move(4)
         def[name] := buildalt(tab(0))
         }
end

procedure buildalt(s)
   local k
   k := []
   every put(k,buildsub(genalt(s)))
   return k
end

procedure buildsub(s)
   local k
   k := []
   every put(k,gensub(s))
   return k
end

procedure genalt(s)
   local t
   s ? while t := tab(upto('|') | 0) do {
      suspend t
      move(1) | break
      }
end
```

```
procedure gensub(s)
   local t
   s ? repeat {
      t := tab(upto('<') | 0)
      if t == "" then {
         move(1) | break
         t := nonterm(tab(upto('>')))
         move(1)
         }
      suspend t
      }
end

procedure generate(s)
   local name, count
   s ? {
      ="<" | fail
      name := tab(upto('>')) | fail
      move(1)
      count := integer(tab(0)) | fail
      }
   every 1 to count do write(synthesize(name))
   return
end

procedure synthesize(s)
   local sentence, nexts, t, x
   sentence := ""
   nexts := [nonterm(s)]
   while t := get(nexts) do
      if type(t) == "nonterm" then {
         x := \def[t.ntname] | {write("*** <",t.ntname,"> undefined"), fail}
         nexts := ?x ||| nexts
         }
      else sentence ||:= t
   return sentence
end
```

Comments The analysis of the grammatical specifications illustrates moderately complicated string scanning. In the scanning expressions, terminators are appended so that successive items can be handled uniformly. Note that **genalt** and **gensub** generate values for **buildalt** and **buildsub**, respectively. This organization of the analysis activities is not necessary, but it partitions logically distinct activities and allows the program to be adapted to other uses by changing the definitions of **buildalt** and **buildsub**. See the exercises.

Exercises

1. Provide a way for allowing the metalinguistic characters |, <, and > to be included in grammars.
2. Using the preceding extension, write a grammar that generates random grammars.
3. Recursive grammars such as those that describe arithmetic expressions, tend to lead to endless growth of the generation list. Provide a mechanism for biasing the selection of alternatives to mitigate this problem.
4. Some kinds of context sensitivity are easily added to the program above. Explore such possibilities.
5. Modify the program above to generate recognition procedures.

Acknowledgments

The Icon programming language was designed by the authors in collaboration with Dave Hanson and Tim Korb. Many other persons too numerous to list here have provided criticism and suggestions that have been incorporated in the current version of the language.

References

1. Farber, David J., Ralph E. Griswold, and Ivan P. Polonsky, "SNOBOL, A String Manipulation Language", *Journal of the ACM*, Vol. 11, No. 1 (January 1964), pp. 21-30.

2. Farber, David J., Ralph E. Griswold, and Ivan P. Polonsky, *SNOBOL 2*, Technical report, Bell Labs, Holmdel, New Jersey, April 1964.

3. Farber, David J., Ralph E. Griswold, and Ivan P. Polonsky, "The SNOBOL 3 Programming Language", *The Bell System Technical Journal*, Vol. XLV, No. 6 (July-August 1966), pp. 895-944.

4. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky, *The SNOBOL4 Programming Language*, second edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971.

5. Griswold, Ralph E., *Bibliography of Documents Related to the SNOBOL Languages*, Technical Report TR 78-18a, Department of Computer Science, The University of Arizona, Tucson, Arizona, September 1979.

6. Griswold, Ralph E., and David R. Hanson, "An Overview of SL5", *SIGPLAN Notices*, Vol. 12, No. 4 (April 1977), pp. 40-50.

7. Hanson, David R., and Ralph E. Griswold, "The SL5 Procedure Mechanism", *Communications of the ACM*, Vol. 21, No. 5 (May 1978), pp. 392-400.

8. Griswold, Ralph E., "String Analysis and Synthesis in SL5", *Proceedings of the ACM Annual Conference*, October 1976, pp. 410-414.

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

10. Kernighan, Brian W., and M. D. McIlroy, *UNIX Programmer's Manual, Seventh Edition*, Bell Laboratories, Murray Hill, New Jersey, January 1979.

11. American National Standards Institute, *USA Standard Code for Information Interchange*, X3.4-1977, New York, New York, 1977.

12. Coutant, Cary A., and Stephen B. Wampler, *A Tour Through The C Implementation of Icon, Version 5*, Technical Report TR 81-11a, Department of Computer Science, The University of Arizona, Tucson, Arizona, December 1981.

13. Gimpel, James F., *Algorithms in SNOBOL4*, John Wiley & Sons, New York, New York, 1976, pp. 25-26.

14. Gimpel, James F., and William Keister, *Minimal Meandering Strings*, Technical report, Bell Labs, Holmdel, New Jersey, July 1970.

15. Wetherell, Charles, *Etudes for Programmers*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978, pp. 30-31.

16. Gimpel, James F., *Algorithms in SNOBOL4*, John Wiley & Sons, New York, New York, 1976, pp. 253-273.

17. Griswold, Ralph E., "Programming Techniques Using Character Sets and Character Set Mappings in Icon", *The Computer Journal*, Vol. 23, No. 2 (May 1980), pp. 107-114.

18. Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976, pp. 143-147.

19. Hanson, David R., "A Procedure Mechanism for Backtrack Programming", *Proceedings of the ACM Annual Conference*, October 1976, pp. 401-405.

20. Griswold, Ralph E., and David R. Hanson, "An Alternative to the Use of Patterns in String Processing", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2 (April 1980), pp. 153-172.

21. Griswold, Ralph E., *String and List Processing in SNOBOL4: Techniques and Applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975, pp. 192-200.

Appendix A

Syntax

Formal Syntax

The following formal syntax for Icon describes only macroscopic features. Complete lists of operators and keywords are included in Appendix B. See Section 2.2.1 for a description of identifiers and Sections 4.1.1, 4.2.1, 5.2.1, and 5.3 for a description of literals. Record types are context sensitive; see Section 8.3. See Chapter 12 for equivalence of characters, situations in which semicolons may be omitted, the continuation of string literals over line terminations, and the treatment of blanks.

The syntactic types *period*, *left-bracket*, and *right-bracket* indicate occurrences of the characters . , [, and] , which have metalinguistic uses in the syntax description language.

program ::= *declaration* ...

declaration ::= *global-declaration* | *external-declaration* | *record-declaration* | *procedure-declaration*

global-declaration ::= global *identifier-list*

identifier-list ::= *identifier* [, *identifier*] ...

external-declaration ::= external *identifier-list*

record-declaration ::= record *identifier* ([*identifier-list*])

procedure-declaration ::= *procedure-header* ; [*local-declaration* ;] ... [*initial-clause* ;] [*procedure-body* ;] end

procedure-header ::= procedure *identifier* ([*identifier-list*])

local-declaration ::= *local-specification identifier-list*

local-specification ::= local | static | dynamic

initial-clause ::= initial *expr*

procedure-body ::= *optexpr* [; *optexpr*] ...

optexpr ::= [*expr*]

expr ::= *literal* | *identifier* | *keyword* | *operation* | *call* | *reference* | *substring* | *list* | *record-object* | *control-struct* | *return* | *compound-expr* | (*optexpr*)

literal ::= *integer-literal* | *real-literal* | *quoted-literal*

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

call ::= *expr* (*expr-list*)

expr-list ::= *optexpr* [, *optexpr*]

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

substring ::= *expr left-bracket expr range expr right-bracket*

range ::= : | +: | −:

list ::= *left-bracket optexpr right-bracket*

record-object ::= *record-type* (*expr-list*)

control-struct ::= *if-then-else* | *while-do* | *until-do* | *every-do* | *repeat* | *case* | *not* | *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*

case ::= case *expr* of { *case-clause* [, *case-clause*] }

case-clause ::= *expr* : *expr* | default : *expr*

not ::= not *expr*

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

next ::= next

break ::= break *optexpr*

return ::= return *optexpr* | suspend *optexpr* | fail

compound-expr ::= { *optexpr* [, *optexpr*] }

Precedence and Associativity

The relative precedence of control structures, operators, and expression-list delimiters arranged in ascending order, follows. For infix operators, the associativity is listed also

	precedence	*type*	*associativity*
if-then-else	1		
while-do	1		
until-do	1		
every-do	1		
repeat	1		
case	1		
break	1		
return	1		
suspend	1		
fail	1		
&	2	infix	left
?	3	infix	left
:=	4	infix	right
<-	4	infix	right
:=:	4	infix	right
<->	4	infix	right
&:=	4	infix	right
+:=	4	infix	right
-:=	4	infix	right
*:=	4	infix	right
/:=	4	infix	right
%:=	4	infix	right
^:=	4	infix	right
==:=	4	infix	right
>=:=	4	infix	right
>:=	4	infix	right
<=:=	4	infix	right
<:=	4	infix	right
~==:=	4	infix	right
||:=	4	infix	right
===:=	4	infix	right
>>=:=	4	infix	right
>>:=	4	infix	right
<<=:=	4	infix	right
<<:=	4	infix	right
~===:=	4	infix	right
?:=	4	infix	right
++:=	4	infix	right
--:=	4	infix	right
**:=	4	infix	right
|||:=	4	infix	right
===:=	4	infix	right
~===:=	4	infix	right
to-by	5		
|	6	infix	left
==	7	infix	left
~==	7	infix	left
<	7	infix	left
<=	7	infix	left

>	7	infix	left
>=	7	infix	left
==	7	infix	left
~==	7	infix	left
>>	7	infix	left
>>=	7	infix	left
<<	7	infix	left
<<=	7	infix	left
===	7	infix	left
~===	7	infix	left
||	8	infix	left
|||	8	infix	left
+	9	infix	left
-	9	infix	left
++	9	infix	left
--	9	infix	left
*	10	infix	left
/	10	infix	left
%	10	infix	left
**	10	infix	left
^	11	infix	right
\	12	infix	left
not	13	prefix	
!	13	prefix	
~	13	prefix	
&	13	prefix	
*	13	prefix	
?	13	prefix	
.	13	prefix	
+	13	prefix	
-	13	prefix	
=	13	prefix	
|	13	prefix	
/	13	prefix	
\	13	prefix	
expr( )	14		
expr[ ]	14		
.	15	infix	left

Reserved Words

The following are reserved words, which cannot be used as identifiers

break	by	case	default	do
dynamic	else	end	every	external
fail	global	if	initial	local
next	not	of	procedure	record
repeat	return	static	suspend	then
to	until	while		

Appendix B

Built-In Operations

The following sections list the built-in operations of Icon, with citations to primary section references

Functions

function	*section*
abs(n)	4.1.2
any(c,s,i,j)	6.3.2
bal(c1,c2,c3,s,i,j)	6.3.2
center(s1,i,s2)	6.1.3
close(f)	9.1
copy(x)	10.4
cset(x)	5.3
display(i,f)	11.4
exit(i)	14.5
find(s1,s2,i,j)	6.3.1
get(a)	8.1.4
image(x)	10.9
integer(x)	4.4.1
left(s1,i,s2)	6.1.3
list(i,x)	8.1.1
many(c,s,i,j)	6.3.2
map(s1,s2,s3)	6.1.5
match(s1,s2,i,j)	6.3.1
move(i)	7.2
numeric(n)	4.5
open(s1,s2)	9.1
pop(a)	8.1.3
pos(i)	7.1
pull(a)	8.1.4
push(a,x)	8.1.3
put(a,x)	8.1.4
read(f)	9.3
reads(f,i)	9.3
real(x)	4.4.2
repl(s,i)	6.1.2
reverse(s)	6.1.5
right(s1,i,s2)	6.1.3
sort(x,i)	8.4
stop(x1, ,xn)	14.5
string(x)	5.4.1
system(s)	10.10
tab(i)	7.2
table(x)	8.2.1
trim(s,c)	6.1.5
type(x)	10.8
upto(c,s,i,j)	6.3.2

write(x1,...,xn)	9.2
writes(x1,...,xn)	9.2

Infix Operators

operator	*section*
:=	2.2.1
<-	3.5
:=:	2.2.1
<->	3.5
&:=	10.2
+:=	10.2
-:=	10.2
*:=	10.2
/:=	10.2
%:=	10.2
^:=	10.2
=:=	10.2
>=:=	10.2
>:=	10.2
<=:=	10.2
<:=	10.2
~=:=	10.2
||:=	10.2
==:=	10.2
>>=:=	10.2
>>:=	10.2
<<=:=	10.2
<<:=	10.2
~==:=	10.2
?:=	10.2
++:=	10.2
--:=	10.2
**:=	10.2
|||:=	10.2
===:=	10.2
~===:=	10.2
&	3.3
+	4.1.2
-	4.1.2
*	4.1.2
/	4.1.2
^	4.1.2
%	4.1.2
=	4.1.3
~=	4.1.3
>	4.1.3
>=	4.1.3
<	4.1.3
<=	4.1.3
++	5.3
--	5.3
**	5.3

||	6 1 1
|||	8 1.5
==	6.2
~==	6.2
>>	6.2
>>=	6.2
<<	6.2
<<=	6.2
===	10.3
~===	10 3
.	8 3 3

Prefix Operators

operator	*section*
+	4 1 2
–	4 1.2
~	5 3
!	10.1
=	7.3
*	2.2.4
&	2 2 2
?	10.5
/	10.7
\	10 7
.	2 3 1

Keywords

keyword	*section*
&ascii	5.3
&clock	10.6
&cset	5.3
&date	10.6
&dateline	10.6
&errout	9 1
&fail	2 3 2
&host	10 11
&input	9.1
&lcase	5.3
&level	11 3 3
&null	10 7
&output	9.1
&pos	7.1
&random	10.5
&subject	7.1
&time	10 6
&trace	11 3 4
&ucase	5 3
&version	10 11

Appendix C

Summary of Defaults

Omitted Expressions

Omitted expressions default to •. For example

```
break
```

is equivalent to

```
break &null
```

Similarly, omitted arguments in function and procedure calls default to •. For example left(s,i) is equivalent to left(s,i,&null) In some functions, null-valued arguments default to commonly used values. These defaults apply whether the argument is explicitly omitted or whether evaluation of the expression given for the argument produces • For example, left(s,i) and left(s,i,&null) are equivalent so far as interpretation of the third argument is concerned Defaults for null-valued arguments are listed below Arguments that are not shown as • are assumed to be non-null Note that for the string analysis functions, the default for the initial position depends on whether the argument specifying the string being examined is •. In all other cases, the default for a null-valued argument is independent of the values of the other arguments.

abbreviated form	*equivalent expression*
any(c,•,•,•)	any(c,&subject,&pos,0)
any(c,s,•,•)	any(c,s,1,0)
bal(•,•,•,•,•,•)	bal(&cset,'(',')',&subject,&pos,0)
bal(•,•,•,s,•,•)	bal(&cset,'(',')',s,1,0)
center(s,i,•)	center(s,i,"□")
display(•,•)	display(&level,&errout)
find(s,•,•)	find(s,&subject,&pos,0)
find(s1,s2,•,•)	find(s1,s2,1,0)
left(s,i,•)	left(s,i,"□")
list(•)	list(0)
many(c,•,•,•)	many(c,&subject,&pos,0)
many(c,s,•,•)	many(c,s,1,0)
map(s,•,•)	map(s,&ucase,&lcase)
match(s,•,•,•)	match(s,&subject,&pos,0)
match(s1,s2,•,•)	match(s1,s2,1,0)
open(s,•)	open(s,"r")
read(•)	read(&input)
reads(•,•)	reads(&input,1)
right(s,i,•)	right(s,i,"□")
sort(x,•)	sort(x,1)
stop(...,•,...)	stop(...,"",...)
trim(s)	trim(s,'□')
upto(c,•,•,•)	upto(c,&subject,&pos,0)
upto(c,s,•,•)	upto(c,s,1,0)
write(...,•,...)	write(...,"",...)
writes(...,•,...)	writes(...,"",...)

Appendix D

Summary of Error Messages

Translator Error Messages

Messages that may occur during translation because of syntax errors in the program are listed below. The translator continues following detection of an error, but the translated program cannot be executed.

end-of-file expected
global, record, or procedure declaration expected
inconsistent redeclaration
invalid argument list
invalid by clause
invalid case clause
invalid case control expression
invalid character
invalid context for break
invalid context for next
invalid context for return or fail
invalid context for suspend
invalid create expression
invalid declaration
invalid default clause
invalid digit in integer literal
invalid do clause
invalid else clause
invalid every control expression
invalid field name
invalid global declaration
invalid if control expression
invalid initial expression
invalid integer literal
invalid keyword
invalid keyword construction
invalid local declaration
invalid operand
invalid operand for unary operator
invalid operand in alternation
invalid operand in assignment
invalid operand in augmented assignment
invalid radix for integer literal
invalid real literal
invalid reference or subscript
invalid repeat expression
invalid section
invalid then clause
invalid to clause
invalid until control expression
invalid while control expression
missing argument list in procedure declaration
missing colon
missing end

```
missing field list in record declaration
missing identifier
missing left brace
missing of
missing procedure name
missing record name
missing right brace
missing right bracket
missing right parenthesis
missing semicolon
missing semicolon or operator
missing then
more than one default clause
unclosed quote
unexpected end of file
```

Translation may be terminated because of various kinds of overflow:

```
out of global symbol table space
out of local symbol table space
out of string space
out of constant table space
out of tree space
yacc stack overflow
```

There is one warning message issued by the translator:

```
redeclared identifier
```

Unlike the messages above, this warning does not prevent the use of the translated program.

Linker Error Messages

There are two programming errors that are detected by the linker:

```
inconsistent redeclaration
invalid field name
```

These errors prevent the program from being run. There is also a way to request the linker to detect identifiers that have not been declared. The message produced is

```
undeclared identifier
```

This message is only a warning, it does not prevent the use of the linked program

Errors During Loading

Errors that occur during loading are issued by the loader, which is not part of the Icon system itself. Errors may occur because insufficient memory is available or because of errors in external procedures (for example, unresolved references). In the case of loader errors, attempts to execute the resulting program may cause a bus error or other malfunction.

Program Error Messages

Program errors are divided into several major categories, depending on the nature of the error.

Category 1: Invalid Type or Form

101	integer expected
102	numeric expected
103	string expected
104	cset expected
105	file expected
106	procedure or integer expected
107	record expected
108	list expected
109	string or file expected
110	string or list expected
111	variable expected
112	invalid type to size operation
113	invalid type to random operation
114	invalid type to subscript operation
115	list or table expected
116	invalid type to element generator
117	missing main procdure

Category 2: Invalid Argument or Computation

201	division by zero
202	remaindering by zero
203	integer overflow
204	real overflow underflow, or division by zero
205	value out of range
206	negative first operand to real exponentiation
207	invalid field name
208	second and third arguments to map of unequal length
209	invalid second argument to open
210	argument to system function too long
211	by clause equal to zero
212	attempt to read file not open for reading
213	attempt to write file not open for writing

Category 3: Capacity Exceeded

301	insufficient storage in heap
302	insufficient storage in string space
303	insufficient storage for garbage collection
304	insufficient storage for system stack

INDEX