# Reference Manual for the Icon Programming Language*

Version 3

(C Implementation for UNIX†)

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

TR 80-2

May 1980

Department of Computer Science
The University of Arizona

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

†UNIX is a trademark of Bell Laboratories.

CONTENTS

Chapter 4 String Processing



Chapter 5 — Structures



Chapter 6 — Input and Output

Chapter 7 Miscellaneous Operations



Chapter 8 Procedures



Chapter 9 Program Preparation



Chapter 10 Programming Considerations



Chapter 11 Running Icon Programs



Chapter 12 Sample Programs

# 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.

SNOBOL2 [2], which was in use for only a short period of time, was a minor refinement of SNOBOL. SNOBOL3 [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. SNOBOL3 retained the single string data type, static pattern matching, and primitive control structures of the original SNOBOL. SNOBOL3 is still in limited use.

SNOBOL4 [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 ablity to produce and process structures. Tables, at the same time, provided a facility for associative reference of a more disciplined type than the indirect reference facility of SNOBOL, although the latter was retained in SNOBOL4. An exotic feature, originally planned for SNOBOL, was realized for the first time in SNOBOL4: runtime compilation allowed strings to be converted into executable code in the course of program execution. Despite the advances in facility, SNOBOL4 retained the primitive control structures of the earlier languages. Because of new data types and operations, SNOBOL4 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. SNOBOL4 is in wide use at the present time for a variety of appplications [5].

SL5 ('SNOBOL Language 5') [6] was an even more radical departure from the earlier languages. SL5 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 SL5 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. SL5 also has a repertoire of elementary string processing operations that are curiously lacking in the earlier languages. The distribution of SL5 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 SL5 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 SL5 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 runtime 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 3 of the Icon programming language. In particular, it refers to the language as 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.

Variables are 'untyped' as in SNOBOL4 and SL5. Thus a variable may have values of any type. Runtime 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.

*UNIX is a trademark of Bell Laboratories.

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., size) with reserved words given in boldface (e.g., **procedure**). There is, of course, no distinction between reserved words and other program text as it is prepared for actual programs, except for the significance of the reserved word names.

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*.

## 1.5 Organization of the Manual

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

The examples are not intended to motivate 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 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 E.

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

# CHAPTER 2

# Basic Concepts

## 2.1 Types

Icon supports several kinds of data, called *types*:

| | |
|---|---|
| integer | procedure |
| real | list |
| string | table |
| cset | stack |
| file | null |

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, tables, and stacks 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 identity object for several operations and it is convertible to other types. For example, the integer equivalent of ● is 0, while the string equivalent of ● is the empty string containing no characters. In addition to the types listed above, there is a facility for defining record types.

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

Integers, real numbers, and strings 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 quotation marks, as in "summary". See Sections 3.1.1, 3.2.1, and 4.2.1 for further descriptions of the methods available for representing literals. Values of types other than these can be constructed and computed in a variety of ways, but they do not have literal representations.

## 2.2 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. Reserved words may not be used as identifiers.

*syntactically correct identifiers*

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

*syntactically erroneous identifiers*

```
23K
report$
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 4.5.4, 5.1.2, 5.2.2, 5.3.2, 5.4.3, and 8.3.2.

One of the most fundamental operations is the assignment of a value to a variable. This operation is performed by the := infix operator. For example, x := 3 assigns the integer value 3 to the identifier x.

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

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, x := z assigns the value of the identifier z to the identifier x.

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

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

> *Note:* The exchange operator associates to the right and returns its left operand as a variable.
>
> *Error Condition:* If the expression on either side of the exchange operation is not a variable, Error 121 occurs.

### 2.2.2 Keywords

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

A keyword is composed of an ampersand ( & ) followed by one of a number of identifiers that have special meanings. Typical keywords are &date, whose value is the current date, and &null, whose value is ●.

Some keywords are variables, and values can be assigned to them to set the status of conditions. An example is &trace, which controls the tracing of procedure calls (see Section 8.3.4). 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 121 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, size(x) produces the size of object x, map(s1,s2,s3) produces a character mapping on s1, and write(s) writes the value of s.

As indicated, 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 10*n* occurs, where *n* is a digit that identifies the expected type. See Appendix E.

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

*Note:* If more arguments are provided 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. An example of a prefix operator is -i, which produces the negative of i. 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 x := y, s1 || s2 (which produces the concatenation of the strings s1 and s2), and s1 == s2 (which compares strings s1 and s2 for equality).

Blanks or 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. 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 i:=j:=k is equivalent to i:=(j:=k). 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, expressions 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 := size(x))
```

Here the second argument of f is an expression that changes the value of x. The effect is as if f(size(x),size(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 := size(x))
```

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

### 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 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 satisified.

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 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.

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

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 ●.

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 ●.

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. A case clause has 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 evaluation of the **case** control structure is terminated. If no comparison succeeds, the expression in the default case clause, if present, is evaluated. The outcome of **case** is the outcome of the selected *expr2*.

> *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 or if no case clause if selected.

An example of a case expression is

```
case size(s1) of {
   1:          m := 0
   size(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 until it fails. The outcome of **repeat** is ●.

6. The control structure

*expr* **fails**

produces ● if *expr* fails and fails if *expr* succeeds. For example,

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

is equivalent to

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

## 2.5 Compound Expressions

Expressions may be compounded to allow a sequence of expressions to appear in a control structure 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 9.2.

## 2.6 Generators

One of the unusual aspects of Icon is the concept of generators. Some expressions are capable of producing a sequence of results if the expression in which they are contained would otherwise fail to produce a result.

The most fundamental generator is alternation

*expr1* | *expr2*

This exression first evaluates *expr1*. If *expr1* succeeds, its result becomes the result of the alternation expression. If *expr1* fails, however, the outcome of the alternation expression is the outcome of evaluating *expr2*. For example,

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

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

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

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

succeeds if x is equal to 1 or 3.

Another generator is

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

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

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

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

*Error Condition:* If the value produced by *expr3* is 0, Error 231 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.

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

The control structure

**every** *expr1* [ **do** *expr2* ]

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

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

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

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

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

The outcome of **every-do** is ●.

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

## 2.7 Goal-Directed Evaluation

Goal-directed evaluation, in which generators produce alternative values in order to obtain a result for an expression, is implicit in the examples given in the preceding section. The term 'backtracking' is used to describe the situation in which evaluation of an expression that has previously produced a result is resumed to obtain an alternative result.

Backtracking occurs in the evaluation of operands of operators and in the evalution of arguments of functions. For example, in

*expr1* + *expr2*

*expr1* is evaluated first. If it fails, the addition operation fails. If it succeeds, *expr2* is evaluated. If *expr2* fails, however, the addition operation does not necessarily fail. Instead, backtracking occurs and an alternative value of *expr1* is sought. If such an alternative 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, alternatives 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, 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.

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*.

## 2.8 The Extent of Backtracking

Backtracking is strictly limited in its extent by syntactic constructions in the program. The extent of backtracking therefore may 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).

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 braces surrounding a sequence of expressions also provide a barrier to backtracking. While failure of *expr2* in

*expr1* & *expr2*

results in backtracking into *expr1*, no such backtracking occurs in

{*expr1*} & *expr2*

With the exception of *expr1* in

**every** *expr1* **do** *expr2*

once an expression in any control structure has produced a result, its evaluation is complete and no backtracking may occur into it. For example, in

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

once *expr1* has either succeeded or failed, no condition (such as the failure of *expr2* or *expr3*) can cause backtracking into *expr1*.

## 2.9 The Reversal of Effects

As described above, backtracking to an earlier point in a computation may take place in order to obtain alternatives of previously evaluated expressions. There is, however, no implicit reversal of effects such as assignments. For example, in the expression

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

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

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

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

```
x := 0; (x <- 1 to 10) & (x > y)
```

if the value of y is 20, the value of x is restored to 0 when the conjunction fails.

*Note:* The reversible assignment operator associates to the right and returns its left operand as a variable.

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

2. The infix operator x <-> y exchanges the values of x and y, but restores the former values if backtracking causes failure in the expression in which the reversible exchange occurred.

*Note:* The reversible exchange operator associates to the right and returns its left operand as a variable.

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

## 2.10 Loop Control

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

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

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

## 2.11 Procedures

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

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

The procedure name identifies the procedure in the same way that 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(size(s1),size(s2))
```

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

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

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

# CHAPTER 3

# Arithmetic

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

## 3.1 Integers

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

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

### 3.1.1 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.

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

*Examples:*

| *expression* | *value* |
|---|---|
| 2r11 | 3 |
| 8r10 | 8 |
| 10r10 | 10 |
| 16rff | 255 |
| 36rcat | 15,941 |

### 3.1.2 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 three arithmetic prefix operators: +, -, and |. +i and -i are equivalent to 0 + i and 0 - i, respectively. That is, -i is the negative of i. |i produces the absolute value of i.

*Examples:*

| *expression* | *value* |
|---|---|
| +100 | 100 |
| -100 | -100 |
| +0 | 0 |
| -0 | 0 |
| -(4 - 700) | 696 |
| \|1 | 1 |
| \|-1 | 1 |

### 3.1.3 Integer Comparison

There are six operations for comparing the magnitude of integers.

| | |
|---|---|
| i = j | equal to |
| i `= j | not equal to |
| i > j | greater than |
| i >= j | greater than or equal to |
| i < j | less than |
| i <= j | less than or equal to |

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

## 3.2 Real Numbers

Real numbers are represented in floating-point format.

*Note:* Floating-point numbers are double precision.

### 3.2.1 Literal Real Numbers

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

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

*Examples:*

| *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 |

### 3.2.2 Real Arithmetic

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

*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 | -1.0 |
| 1.0 * 2.0 / 2 | 1.0 |
| 1.0 / 2.0 * 2 | 1.0 |
| 4.7 % 2.0 | 0.7 |
| 2.5 % 1.0 | 0.5 |
| +1.0 | 1.0 |
| -1.0 | -1.0 |
| \|-1.0 | 1.0 |

### 3.2.3 Comparison of Real Numbers

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

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

*Examples:*

| *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* |

## 3.3 Mixed-Mode Arithmetic

Except for exponentiation, if either operand of an infix operation is a real number, the other operand is converted to 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* |
|---|---|
| 1.0 + 2 | 3.0 |
| 1 + 2.0 | 3.0 |
| 1 - 2.0 | -1.0 |
| 1.0 * 2 | 2.0 |
| 1.0 / 2 | 0.5 |
| 2 / 1.0 | 2.0 |
| 1 - 1 - 1.0 | -1.0 |
| 1 * 2.0 / 2 | 1.0 |
| 1 / 2.0 * 2 | 1.0 |
| 1.0 / 2 * 2 | 1.0 |
| 2.0 ^ 2 | 4.0 |
| 2.0 ^ -1 | 0.5 |

## 3.4 Arithmetic Type Conversion

### 3.4.1 Conversion to Integer

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

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.

If the string corresponds to a real literal, real-to-integer conversion is performed. See Section 3.4.2. The empty string is converted to the integer 0. See Section 4.2.2.

*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 converted to strings and then to integers. See Section 4.4.
5. The integer equivalent of ● is 0.

*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, csets, and ●.

*Error Condition:* If the conversion fails, Error 101 occurs.

*Examples:*

| *expression* | *value* |
|---|---|
| 1 + "10" | 11 |
| 2^ 4.0 | 16.0 |
| 1 > &null | 0 |

### 3.4.2 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, cset, or ●.

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. The empty string is converted to 0.0. See Section 4.2.2.

*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 4.4.
5. The real number equivalent to ● is 0.0.

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

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

*Error Condition:* If conversion fails, Error 102 occurs.

*Examples:*

| *expression* | *value* |
|---|---|
| 1.0 + "10.0" | 11.0 |
| "2.0" ^ 3 | 8.0 |
| 1.0 > &null | 0.0 |

## 3.5 Numeric Test

The function numeric(x) returns the integer or real number corresponding to x if x is an integer, real number, or if it is convertible to one of these types. See Section 3.4. 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("36rcat") | 15,941 |
| numeric("3r4") | *none* |
| numeric("") | 0 |
| numeric(&null) | 0 |

# CHAPTER 4

# String Processing

## 4.1 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 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 4.6) and for sorting (see Section 5.6). The full set of 256 characters similarly are represented by octal codes 000 through 377 (hexadecimal codes 00 through FF).

## 4.2 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 Section 4.5.

### 4.2.1 Literal Strings

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

*Examples:*

| *expression* | *value* |
|---|---|
| "X" | X |
| 'X' | X |
| "□" | □ |
| "abcd" | abcd |
| "Isn't□it□great?" | Isn't□it□great? |
| '"whoopee".' | "whoopee". |

> *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 | \\ |
| left brace ( { ) | \< |
| right brace ( } ) | \> |
| left bracket ( [ ) | \( |
| right bracket ( ] ) | \) |
| *octal code* | *ddd* |
| *hexadecimal code* | \x*dd* |

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. 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 |
| "\\" | \ |

### 4.2.2 String Size

The size of a string is the number of characters it contains and is computed by size(s). 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* |
|---|---|
| size("abcd") | 4 |
| size("□") | 1 |
| size("") | 0 |

## 4.3 Character Sets

Whereas a string is an ordered sequence of characters in which the same character may appear more than once, a character set (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 4.7.2 and 4.8.3. 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 121 occurs.

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

*Examples:*

| *expression* | *value* |
|---|---|
| cset("abcd") | a b c d |
| cset("badc") | a b c d |
| cset("energy") | e g n r y |

There are four operations on character sets:

1. The value of ~c is the complement of c with respect to &cset.
2. The value of c1 ++ c2 is the union of c1 and c2.
3. The value of c1 ** c2 is the intersection of c1 and c2.
4. The value of c1 -- c2 is the difference of c1 and c2; that is, all of the characters in c1 that are not in c2.

*Examples:*

| *expression* | *value* |
|---|---|
| c1 := cset("drama") | a d m r |
| c2 := cset("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 |

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

## 4.4 Type Conversion

### 4.4.1 Explicit Conversion

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

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 4.6).

*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 |

4. For ●, the result is ■.

*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, cset, or ●. If x is not a string, 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(1088) | 0 1 8 |
| cset(3.14) | . 1 3 4 |

### 4.4.2 Implicit Conversion

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

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

*Examples:*

| *expression* | *value* |
|---|---|
| size(010) | 2 |
| size(10) | 2 |
| size(&null) | 0 |

For operations that require csets, implicit conversion is performed automatically for integers, reals, strings, and ●. The conversion is performed by first converting to a string, if necessary, and then to a cset.

*Error Condition:* If an object of any other type is encountered in a context that requires implicit conversion to a cset, Error 105 occurs.

## 4.5 Constructing Strings

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

### 4.5.1 Concatenation

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

*Note:* The 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" \|\| &null | abcd |
| "" \|\| "" | ■ |

### 4.5.2 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) | ■ |

### 4.5.3 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:* An omitted value of 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:* An omitted value of 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 one character to the left of center.

*Default:* An omitted value of 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 |

### 4.5.4 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 CAPITAL are

```
C A P I T A L
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
1 2 3 4 5 6 7 8
```

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:

```
C A P I T A L
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
-7 -6 -5 -4 -3 -2 -1 0
```

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

The positions that can be specified for a string s are in the range -size(s) to size(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-(size(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 121 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 |

### 4.5.5 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:* An omitted value of c defaults to cset("□").

*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.

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

*Defaults:* An omitted value of s2 defaults to &ucase and an omitted value of 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("abcda","bcad","1234") | 31243 |
| map("abcda","abac","1234") | 324d3 |
| map("wxyz","zyxw","abcd") | dcba |

## 4.6 String Comparison

Strings, like numbers, can be compared, but the basis for comparison is lexical (alphabetical) order rather than numerical value. Lexical order includes all characters 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 function llt(s1,s2) succeeds if s1 is lexically less than s2 and fails otherwise. The value returned on success is s2.

*Examples:*

| *expression* | *value* |
|---|---|
| llt("X","x") | x |
| llt("x","X") | *none* |
| llt("x","x") | *none* |
| llt("XX","x") | x |
| llt("xx","xX") | *none* |
| llt("xx","xxx") | xxx |
| llt("xx","xxX") | xxX |
| llt("","x") | x |
| llt("","") | *none* |

In all, there are four lexical comparison predicates and two lexical comparison operators:

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

## 4.7 String Analysis

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

### 4.7.1 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, that is, i+size(s1).

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

*Defaults:* An omitted value of i defaults to 1 and an omitted value of 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:* An omitted value of i defaults to 1 and an omitted value of 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 |

## 4.7.2 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:* An omitted value of i defaults to 1 and an omitted value of 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",1,1) | *none* |

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:* An omitted value of i defaults to 1 and an omitted value of 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:* An omitted value of i defaults to 1 and an omitted value of 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) | *none* |
| 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:* An omitted value of i defaults to 1 and an omitted value of j defaults to 0. An omitted value of c1 defaults to &cset, an omitted value of c2 defaults to cset("("), and an omitted value of c3 defaults to cset(")").

*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 |

## 4.8 String Scanning

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

The control structure

**scan** *expr1* **using** *expr2*

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

*Failure Condition:* If *expr1* fails, *expr2* is not evaluated and **scan-using** fails.

### 4.8.1 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, but assignment to a substring of &subject sets &pos to the position at the end of the replaced substring.

*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 |

### 4.8.2 Positional Analysis

There are two functions that change &pos automatically and return the substring between the previous and new values of &pos. This substring is called a *scanned substring*.

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

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

   *Examples:*

| *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 |

### 4.8.3 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:* These default interpretations apply only if all three of the trailing arguments are omitted.

*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 |

### 4.8.4 String Transformation

The control structure

**transform** *expr1* **using** *expr2*

is similar to the **scan-using** expression, except that the result of evaluating *expr1* must be a variable and the value of &subject upon completion of the evaluation of *expr2* is assigned to *expr1*. The outcome of **transform-using** is the outcome of *expr2*.

*Failure Condition:* If *expr1* fails, *expr2* is not evaluated and **transform-using** fails.

*Error Condition:* If the evaluation of *expr1* does not produce a variable, Error 121 occurs.

Since the value assigned to *expr1* is the value of &subject when evaluation of *expr2* is complete, *expr2* can be used to change &subject to produce a desired transformation.

One way to change &subject is simply to assign a value to it in *expr2*. For example,

```
transform s using &subject := tab(upto(":"))
```

deletes the trailing portion of s starting at the first colon.

Assignment can also be made to tab(i) and move(i) to replace their scanned substrings in &subject. When an assignment to a scanned substring is made, &pos is set to the end of the replaced substring. For example,

```
transform s using
   while tab(upto("□")) do
      tab(many("□")) := "□"
```

replaces all occurrences of multiple blanks in s by single blanks.

Assignment to scanned substrings is a reversible effect. If such an assignment is made, but the expression in which it occurs fails, &subject and &pos are restored to their former values.

*Warning:* Assignment to a scanned substring may change the length of &subject and the value of &pos.

*Notes:* Any form of assignment may be made to a scanned substring: reversible assignment, exchange, and reversible exchange. Assignment may also be made to =s. Since assignment to a scanned substring is a reversible effect, := and :=: are equivalent to <- and <->, respectively, in such contexts.

Insertion into &subject is frequently useful. The function insert(s,i) inserts s into &subject following position i. The value returned by insert(s,i) is s as a scanned substring.

*Failure Condition:* insert fails if i is out of range.

*Default:* An omitted value of i defaults to &pos.

*Note:* insert(s,i) changes the value of &pos to i+size(s).

*Examples:*

| *expression* | *value* | *value of* &subject | value of &pos |
|---|---|---|---|
| &subject := "abcd" | abcd | abcd | 1 |
| insert("x") | x | xabcd | 2 |
| tab(0) | abcd | xabcd | 6 |
| insert("yy",3) | yy | xayybcd | 5 |

### 4.8.5 Nested Scanning

The values of &subject and &pos are saved on entry to **scan-using** and **transform-using** and restored upon exit. Consequently, nested scanning is possible. For example, if words contains a sequence of words followed by blanks, the following expressions

```
twords := ""
scan words using
   while scan tab(upto("□")) using
      if upto("t") then twords := twords || &subject || "□"
   do move(1)
```

assign a similar string to twords, but with only those words containing the letter t.

Similarly, the **transform-using** expression may be used to transform scanned substrings. For example

```
transform s using
   while tab(upto(&lcase)) do
      transform tab(many(&lcase)) using
         &subject := reverse(&subject)
```

reverses all strings of lower-case letters in s.

*Note:* The values of &subject and &pos are not restored if the **using** clause is exited by a **break**, **next**, or a procedure return.

# CHAPTER 5

# 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.

## 5.1 Lists

Lists are sequences of variables that are referenced by position. Lists are equivalent to vectors and one-dimensional arrays.

### 5.1.1 Creation of Lists

Lists are created during program execution by the function list(i,j), where i is the lower bound and j is the upper bound. The initial value of all elements of the list is ●.

> *Defaults:* If j is omitted, the lower bound defaults to 1 and i specifies the upper bound.
>
> *Error Conditions:* If a bound specification is less than $-2^{15}$ or greater than $2^{15}-1$, Error 205 occurs. If the upper bound is less than the lower bound, Error 216 occurs.

The functions lbound(x) and ubound(x) return the lower and upper bounds, respectively, of the list x. The function size(x) gives the number of elements in x.

*Examples:*

| *expression* | *value* |
|---|---|
| dec := list(1,10) | *list* |
| lbound(dec) | 1 |
| ubound(dec) | 10 |
| ndec := list(10) | *list* |
| lbound(ndec) | 1 |
| ubound(ndec) | 10 |
| sector := list(-5,2) | *list* |
| lbound(sector) | -5 |
| ubound(sector) | 2 |
| size(sector) | 8 |

A list also may be created by an expression of the form

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

where the values of the expressions are the initial values of the list elements. In this case, the lower bound is 1 and the upper bound is the number of expressions in the list.

> *Note:* The expressions in the list may be empty. The number of elements is the number of commas plus one.

*Examples:*

| *expression* | *value* |
|---|---|
| triple := [0,0,0] | *list* |
| lbound(triple) | 1 |
| ubound(triple) | 3 |
| line := [,,,] | *list* |
| size(line) | 4 |
| octave := [1,2,3,4,5,6,7,8] | *list* |
| size(octave) | 8 |
| unit := [ ] | *list* |
| size(unit) | 1 |

### 5.1.2 Accessing List Elements

An element of a list is 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 is not between the lower and upper bounds, inclusive. In this case the subscript is said to be *out of range*.

*Examples* (for the lists given in the preceding examples):

| *expression* | *value* |
|---|---|
| dec[3] := 1 | 1 |
| dec[5] := dec[3] * 5 | 5 |
| dec[0] | *none* |
| octave[4] | 4 |
| unit[1] | ● |

### 5.1.3 Open Lists

Lists ordinarily are fixed in size. Lists may be opened for expansion so that they can be subscripted beyond the original upper bound. A list is opened by the expression open(x). Subsequently, x expands automatically when assignment is made to an subscript that is one beyond its current upper bound.

*Notes:* Lists are closed when they are created. Expansion of an open list occurs only when the subscript is one beyond the current upper bound. References to larger subscripts fail. Expansion occurs only when an assignment is actually made. A reference to the value at a position one beyond the current end of an open list produces ●. See also Section 5.5. open(x) modifies x and also returns the modified value.

The function close(x) closes x and prevents x from being expanded by out-of-range references.

*Note:* close(x) modifies x and also returns the modified value.

*Examples:*

| *expression* | *value* |
|---|---|
| laundry := list(10) | *list* |
| size(laundry) | 10 |
| laundry[1] | ● |
| laundry[11] := "shirts" | *none* |
| open(laundry) | *list* |
| laundry[12] := "shirts" | *none* |
| laundry[11] := "shirts" | shirts |
| size(laundry) | 11 |
| laundry[12] := "socks" | socks |
| size(laundry) | 12 |
| close(laundry) | *list* |
| size(laundry) | 12 |
| laundry[12] | socks |
| laundry[13] | *none* |

### 5.1.4 The Empty List

The value of list(0) is an empty list containing no elements. Unlike other lists, the empty list is open when it is created.

*Examples:*

| *expression* | *value* |
|---|---|
| seq := list(0) | *list* |
| size(seq) | 0 |
| seq[1] := 1 | 1 |
| seq[3] | *none* |
| seq[2] := 4 | 4 |
| size(seq) | 2 |

## 5.2 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.

### 5.2.1 Creation of Tables

Tables are created during program execution by the function table(i). When a table is created, it is empty and has no elements. Elements may be added at will and tables grow automatically. The size of the table is limited to the value of i. A size of 0 specifies a table that is not limited in size, except by the amount of storage that is available.

*Default:* An omitted value of i defaults to 0.

*Error Condition:* If i is less than 0 or greater than $2^{15}$-1, Error 205 occurs.

### 5.2.2 Accessing Table Elements

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

*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 7.3.

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. The function size(t) gives the size of the table t.

The value of a table element that is not in the table is ●. Table elements are only created, however, when values are assigned to them. See also Sections 5.2.3 and 5.5.

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

*Examples:*

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

### 5.2.3 Closed Tables

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

*Note:* close(t) modifies t and also returns the modified value.

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

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

*Examples:*

| *expression* | *value* |
|---|---|
| digram := table(50) | *table* |
| digram["th"] := 73 | 73 |
| digram["en"] := 81 | 81 |
| digram["io"] := 41 | 41 |
| close(digram) | *table* |
| digram["th"] := 74 | 74 |
| digram["st"] | *none* |
| open(digram) | *table* |
| digram["st"] | ● |

## 5.3 Stacks

A stack is an aggregate of variables that resembles a list. A stack, however, grows and shrinks automatically as elements are added (pushed) and deleted (popped). Stacks usually are accessed only at the most recently added (top) element.

### 5.3.1 Creation of Stacks

Stacks are created during program execution by the function stack(i). The maximum number of elements the stack may have is limited by i. A size of 0 specifies a stack of unlimited size.

*Default:* An omitted value of i defaults to 0.

*Error Condition:* If i is less than 0 or greater than $2^{15}$-1, Error 205 occurs.

### 5.3.2 Accessing Stacks

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

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

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

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

Stacks also can be referenced by position like lists. k[1] references the top element of the stack k. k[2] references the next element below the top, and so on.

*Failure Conditions:* pop(k) and top(k) fail if k is empty. k[i] fails if i is less than 0 or greater than the size of k.

*Examples:*

| *expression* | *value* |
|---|---|
| pstack := stack(50) | *stack* |
| size(pstack) | 0 |
| push(pstack,"x") | x |
| push(pstack,"y") | y |
| pstack[1] | y |
| pstack[2] | x |
| pstack[3] | *none* |
| push(pstack,"*") | * |
| size(pstack) | 3 |
| top(pstack) | * |
| size(pstack) | 3 |
| pop(pstack) | * |
| size(pstack) | 2 |
| top(pstack) := "z" | z |
| size(pstack) | 2 |
| pop(pstack) | z |
| pop(pstack) | x |
| size(pstack) | 0 |
| pop(pstack) | *none* |
| top(pstack) | *none* |

## 5.4 Records

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

### 5.4.1 Declaring Record Types

A record type is declared in the form

**record** *name* ( [ *identifier* [ , *identifier* ] ... ] )

The name specifies a new type, which is added to the repertoire of types. See Section 7.8. The identifiers 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(r,i)

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

### 5.4.2 Creating Records

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

*type* ( *expr* [ , *expr* ] ... )

where the type is one declared by **record** and the values of the expressions are assigned to 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 r field and a value of 2.5 for the i field.

*Default:* Omitted trailing arguments in a record creation expression default to ●.

The value of size(z) is the number of fields declared for the type of record z.

### 5.4.3 Accessing Records

A record is accessed by field name, using the infix . operator. Continuing the example above, the value of z.r 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.

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

*Failure Condition:* z[i] fails if i is less than 0 or greater than the number of fields in z.

*Examples:*

| *expression* | *value* |
|---|---|
| z1 := complex(0,0) | *complex* |
| z2 := complex(3.14, 2.0) | *complex* |
| z1.r | 0 |
| z1.r + z2.i | 2.0 |
| z1.r := z2.r | 3.14 |
| z2[2] | 2.0 |
| z2[3] | *none* |

## 5.5 Assigning Values to Structure Elements

The infix operator ::= assigns a value to all elements of a structure. For example, if x is a list, x ::= 0 assigns 0 to all elements of x. For lists and tables, the ::= operator also establishes a default value for newly created elements. This default value is the value of an element of an open list one position beyond its current size and the value produced when an element not already in a table is referenced.

*Note:* The structure assignment operator associates to the right and returns its left operand as a variable.

*Error Condition:* If the expression on the left side of the structure assignment operator is not a list, table, stack, or record object, Error 114 occurs.

*Examples:*

| *expression* | *value* |
|---|---|
| line := list(10) ::= 1 | *list* |
| line[1] | 1 |
| line ::= 3 | *list* |
| line[1] | 3 |
| open(line) | *list* |
| line[11] | 3 |
| syns := table(0) | *table* |
| syns["loc"] | ● |
| syns ::= 0 | *list* |
| syns["loc"] | 0 |

## 5.6 Sorting Structures

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

In sorting, strings are sorted in non-decreasing lexical order (see Section 4.6), while integers and real numbers are sorted in non-decreasing numerical order (see Sections 3.1.3 and 3.2.3). 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
- stacks
- 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 (open) list.

*Default:* An omitted value of i defaults to 1.

*Error Conditions:* In sort(x), if x is not a list or a table, Error 219 occurs. In sort(t,i), if i is not 1 or 2, Error 220 occurs.

# CHAPTER 6

# Input and Output

## 6.1 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 121 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. 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 computer terminal where the terminal 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:* An omitted value of s2 defaults to r.
>
> *Notes:* If a file is opened for writing but not reading, create is implied. Create and append have no effect on pipes. Pipes may not be opened simultaneously for 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 221 occurs.

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

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

## 6.2 Writing Data to Files

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

*Notes:* A line terminator is added after sn. 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(f,s1,s2,...,sn) writes s1, s2, ..., sn to file f in the manner of write(f,s1,s2,...,sn), but no line terminator is appended at the end. 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.

*Default:* If the first argument to write or writes is not of a file, the arguments are written to &output. That is, write(s1,s2,...,sn) writes s1, s2, ..., sn to &output.

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

During writing, integers, real numbers, csets, and ● are automatically converted to strings as described in Section 4.4. Arguments of other types are converted to strings by use of the image function (see Section 7.9). Thus arguments of any type can be specified in the write and writes functions.

*Examples:*

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

## 6.3 Reading Data from Files

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

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

*Default:* An omitted value of f defaults to &input.

*Note:* The maximum input line length is 256.

*Error Conditions:* If the argument to read is not a file, Error 106 occurs. If an input line exceeds the maximum length, Error 411 occurs. If an attempt is made to read from a file which is not opened for reading, Error 402 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.

*Default:* An omitted value of f defaults to &output.

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

*Error Conditions:* If the argument to reads is not a file, Error 106 occurs. If i is less than 1, Error 205 occurs. If an attempt is made to read from a file which is not opened for reading, Error 402 occurs.

# CHAPTER 7

# Miscellaneous Operations

## 7.1 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 lower bound to the upper bound. For example, if x is a list

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

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

For tables, the order of generation is unpredictable, but all elements are generated. For stacks, the order of generation is from the top of the stack to the bottom of the stack. 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.

## 7.2 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.

The other augmented assignment operators are:

```
-:=
*:=
/:=
%:=
^:=
||:=
++:=
--:=
**:=
```

*Error Condition:* If the expression on the left side of an augmented assignment operator is not a variable, Error 121 occurs.

## 7.3 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, stacks, 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 5.2.2.

*Examples:*

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

## 7.4 Copying Objects

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

```
x := list(10)
y := x
```

assign the same list to x and y. Subsequently, x[3] and y[3] reference the same element of the same list.

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

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

assigns a copy of x to z. This copy has the same structure as x and the values of all the elements are the same, but x and z are distinct objects. Subsequently, x[3] and z[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 7.3.

## 7.5 Random Number Generation

The value of random(i) is an integer from a pseudo-random sequence with the range 1 to i, inclusive.

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 Conditions:* If the value of i in random(i) is less than one or greater than $2^{15}$-1, Error 205 occurs. If the value assigned to &random is less than zero or greater than $2^{15}$-1, Error 205 occurs.

## 7.6 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 April 1, 1980 is 1980/04/01.

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 Sunday, April 13, 1980 5:16 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 121 occurs.

## 7.7 The Null Value

The null value, ●, is an identity in the concatenation of strings and in the addition of numeric objects. It is also useful to indicate the end of a chain of pointers composed of structure objects.

*Note:* There is only one null value.

The function null(x) converts x to ●. The values convertible to ● are the empty string, the integer 0, the real number 0.0, the empty character set, and ● itself.

*Failure Condition:* null(x) fails if x is not one of these values.

*Examples:*

| *expression* | *value* |
|---|---|
| null("") | ● |
| null("□") | *none* |
| null(0) | ● |
| null("0") | *none* |
| null(0.0) | ● |
| null(&null) | ● |

## 7.8 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(&null) | null |

## 7.9 String Images

The function image(x) produces a string that represents the value of x. For strings, this includes enclosing double quotes and escapes as necessary. File names are enclosed in single quotes to avoid ambiguities with the images of strings. The images of lists include the upper and lower bounds. For other structures, their current size is given.

*Examples:*

| *expression* | *value* |
|---|---|
| image(1) | 1 |
| image(2.0) | 2.0 |
| image('abc') | "abc" |
| image("") | "" |
| image(&null) | &null |
| image(cset("drama")) | cset("admr") |
| image(&input) | &input |
| image(open("data")) | 'data' |
| image([1,0,11]) | list(3) |
| image(list(-3,2)) | list(-3,2) |
| image(trim) | function trim |
| image(complex(3.1,1.0)) | record complex(2) |

## 7.10 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 222 occurs.

## 7.11 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 3, March 28, 1980 .

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

# CHAPTER 8

# Procedures

## 8.1 Procedure Declarations

A procedure 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 9.2.

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
```

## 8.2 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.

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

## 8.3 Procedure Activation

### 8.3.1 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(size(x),size(y))
```

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

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

*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.

### 8.3.2 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 outcome of the procedure call is the outcome of the last expression in the procedure body.

*Failure Condition:* A procedure call fails if the last expression in the procedure body 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 result of evaluating *expr*. If *expr* fails, the procedure call fails. Otherwise the value of *expr* becomes the value of the calling expression. For example

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

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

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 another alternative. Thus suspended procedures are generators. Consider the following procedure.

```
procedure timer(t)
   while &time - t do suspend
   fail
end
```

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

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

evaluates *expr* repeatedly during an interval of approximately 1000 milliseconds.

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 outcome of **suspend** itself, once all alternatives of *expr* have been produced, is ●.

If the expression in **return** 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.

Unlike **return**, **suspend** does no dereference local identifiers, since they remain in existence while the procedure is suspended.

### 8.3.3 Procedure Level

Since procedures can invoke other procedures before they return, several procedures may be invoked at any one time. The value of the &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 504 occurs. &level is not a variable. If an attempt is made to assign a value to it, Error 121 occurs.

### 8.3.4 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.

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 dots equal to the level from which the call is made (&level). In the case of procedure return, the trace message includes the function name, the type of return, and the value returned, except in the case of failure. The indentation corresponds to the level to which the return is made. All trace messages include the name of the file containing the procedure that is traced.

An example is given by the following program:

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

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

The trace output produced by this program is

```
.acker(1,3) called from line 10 in acker.icn
..acker(1,2) called from line 5 in acker.icn
...acker(1,1) called from line 5 in acker.icn
....acker(1,0) called from line 5 in acker.icn
.....acker(0,1) called from line 4 in acker.icn
.....acker returned 2 at line 3 in acker.icn
....acker returned 2 at line 4 in acker.icn
....acker(0,2) called from line 5 in acker.icn
....acker returned 3 at line 3 in acker.icn
...acker returned 3 at line 5 in acker.icn
...acker(0,3) called from line 5 in acker.icn
...acker returned 4 at line 3 in acker.icn
..acker returned 4 at line 5 in acker.icn
..acker(0,4) called from line 5 in acker.icn
..acker returned 5 at line 3 in acker.icn
.acker returned 5 at line 5 in acker.icn
main returned 5 at line 10 in acker.icn
```

## 8.4 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.

*Defaults:* An omitted value of i defaults to 1 (only the identifiers in the currently invoked procedure are displayed). An omitted value of f defaults to &errout.

*Error Condition:* If the second argument to display is not a file, Error 106 occurs.

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

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

```
global hexd

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

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

The output of display(&level) is

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

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

## 8.5 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.

# CHAPTER 9

# Program Preparation

## 9.1 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 evey 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.

## 9.2 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 the Icon syntax 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
```

> *Warning:* Care should be taken not to split expressions at places where components are optional. For example
>
> ```
> return e
> ```
>
> and
>
> ```
> return
> e
> ```
>
> are quite different.

Identifiers may be arbitrarily long, but procedure names must be unique in their first seven characters.

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

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

## 9.3 Program Character Set

Icon uses the ASCII character set [11]. In program text, the following characters are syntactically equivalent:

> lower-case letters and corresponding upper-case letters
> blank and tab

> *Note:* In literal strings, all characters are distinct and the equivalences above do not apply. See Section 4.2.1.

## 9.4 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 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.

## 9.5 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 shows the use of comments.

```
# These procedures print all the intersections of two words.
# xcross 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 size(word2) ],j))
                                                # then rest of word2
end
```

# CHAPTER 10

# Programming Considerations

## 10.1 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 end of 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 conversiions, 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 evey 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 perform an explicit conversion out of line 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 require a comparatively slow lookup process.

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 alternatives (see the eight-queens program in Chapter 12).

## 10.2 Programming Pitfalls

Since Icon has several unusual features, the novice Icon programmer is likely to run into 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 alternatives are attempted 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, 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, stack, 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 x is a list, y := x assigns the *same* list to y. Thus assignment to an element of x changes that element of y. Similarly, the effect of

```
x := list(3) ::= list(5)
```

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

4. Exit from within a **using** clause, 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 within a **using** clause.

5. The outcome of **suspend**, once it has produced all its alternatives, is ●. At the same time, in the absence of a specific return at the end of a procedure body, the outcome of a call of that procedure is the outcome of the last expression in the procedure body. If this last expression is a **suspend**, the outcome of the call is ●. If the procedure is intended to serve as a generator, this final ● may be spurious. This problem can be avoided by placing a **fail** after the **suspend**. See, for example, the sentence recognizer in Chapter 12.

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. 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.

8. Since upper- and lower-case letters are equivalent, except in string literals, identifiers such as VALUE and value are the same, although they may appear to be different. This equivalence may lead to unexpected collisions of identifiers.

9. 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 runtime type error.

# CHAPTER 11

# Running Icon Programs

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

## 11.1 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 E 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 E.

## 11.2 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.

Separately translated program segments may be linked together to form a complete program. Thus useful utility procedures can be kept in a library and linked when needed.

## 11.3 Loading

Once an Icon program is linked, it is loaded ("link edited") by the UNIX program *ld* [10].

The error message text overflow from *ld* indicates that there is not enough memory available to run the Icon program. This problem may be due to the program itself being too large or to its use of too many different operators and functions.

> *Note:* Each different operator and function referenced in a program requires a corresponding runtime module to be loaded.

## 11.4 Program Execution

Program execution is initiated by invoking the procedure main.

If there are any arguments on the UNIX command line used to intitiate 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).

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

*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.

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

## 11.6 Error Termination

Errors during program execution may result from logical mistakes, invalid data, and so forth. If one of these errors occurs, an error number and an explanatory message are printed and program execution is terminated with an exit status of 2. See Appendix E.

# CHAPTER 12

# 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 internal 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.

## Problem 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 [12]. 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.

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

*Exercises:*

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

## Problem 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 [13]. 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 13, 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.

```
procedure meander(alpha,n)
   local result, t, i, c, k
   i := k := size(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.

### Problem 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.

```
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 size(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

```
fish
 c
 h
 o
 o
 l

  s
  c
fish
 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 [14].

### Problem 4: Word Tabulation

*Description:* One of the simplest illustrations of the utility of string scanning, as opposed to more primitive string analysis methods, is tabulation of the words contained in a text file. 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.

```
procedure wordcount(n)
   local t, line, x, y
   static letters
   initial letters := &lcase ++ &ucase
   t := table()
   every line := !&input do
      scan line using
         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.
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.

## Problem 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.

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

procedure tform(s)
   local value,left,right
   if null(s) then return &null
   scan s using
      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)
   if null(t) then fail
   suspend walk(t.ltree | t.rtree)
   return t.data
end

procedure leaves(t)
   if null(t) then fail
   if null(t.ltree) & null(t.rtree) then return t.data
   suspend leaves(t.ltree | t.rtree)
   fail
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.

### Problem 6: Displaying Scanning

*Description:* Provide a procedure to display the state of string scanning.

*Solution:* This problem leaves much leeway for interpretation. Since such a procedure can be a useful diagnostic aid, it should present information in a concise but easily interpreted format. The solution that follows shows &subject together with position numbering bars. &pos is indicated by a pointer to its position in &subject.

```
procedure disp()
   local c, subject
   static bars, nline
   initial {
      bars := repl("| ",50) || "|"
      nline := repl("1 2 3 4 5 6 7 8 9 0 ",5)
      }
   every c := !&subject do subject ||:= " " || c
   write("\n",subject)
   write(bars[1+:2*(&pos-1)],"^",bars[2+:2*(size(&subject)-&pos+1)])
   write(nline[1+:2*size(&subject)+1])
end
```

*Comments:* The display obtained by the use of this procedure is illustrated by an adaptation of the example given in Section 4.8.5:

```
words := "it is a test "
scan words using {
   disp()
   while scan tab(find(" ")) using {
         disp()
         if find("t") then twords ||:= &subject || " "
         }
   do {move(1); disp()}
   }
```

The display output is

```
 i t   i s   a   t e s t
^ | | | | | | | | | | | | |
1 2 3 4 5 6 7 8 9 0 1 2 3 4

 i t
^ | |
1 2 3

 i t   i s   a   t e s t
| | | ^ | | | | | | | | | |
1 2 3 4 5 6 7 8 9 0 1 2 3 4

 i s
^ | |
1 2 3

 i t   i s   a   t e s t
| | | | | | ^ | | | | | | |
1 2 3 4 5 6 7 8 9 0 1 2 3 4

 a
^ |
1 2

 i t   i s   a   t e s t
| | | | | | | ^ | | | | |
1 2 3 4 5 6 7 8 9 0 1 2 3 4

 t e s t
^ | | | |
1 2 3 4 5

 i t   i s   a   t e s t
| | | | | | | | | | | | ^
1 2 3 4 5 6 7 8 9 0 1 2 3 4
```

*Exercises:*

1. The length of &subject that can be handled by this solution is quite limited. Remove the limitation, allowing for multiple lines of output for very long subjects.

2. Improve the format of the output so that the numerical value of &pos can be determined more easily.

3. The **every** loop used to intersperse blanks between the characters of &subject is time consuming and wasteful of storage. Using positional transformation techniques [15,16], replace this loop by a single mapping expression.

## Problem 7: Eight Queens

*Description:* The classic example used to illustrate backtracking is the eight-queens problem [17,18], 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.

```
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)
   fail
end

procedure place(r,c)
   static up, down, rows
   initial {
      up := list(-7,7) ::= 0
      down := list(2,16) ::= 0
      rows := list(1,8) ::= 0
      }
   if 0 = rows[r] = up[r-c] = down[r+c] then
      every rows[r] <- up[r-c] <- down[r+c] <- 1 do
         suspend r
   fail
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.

## Problem 8: 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 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.

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

procedure strip(s)
    local t
    while scan s using ="(" & t := tab(bal(")")) & pos(-1) do
       s := t
    return s
end

procedure lassoc(s,c)
   local j
   j := 0
   scan s using every j := bal(c)
   if j = 0 then fail else return form(s,j)
end

procedure rassoc(s,c)
   local j
   scan s using j := bal(c) | fail
   return form(s,j)
end

procedure form(s,k)
   local a1, a2, op
   scan s using {
      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 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.

## Problem 9: 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 [19]. 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
```

Recognition procedures s and t corresponding to <s> and <t> are

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

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

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

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

Thus 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.

A procedure to control the recognition process specifies the goal and applies it to the text in question:

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

The procedure recogn succeeds or fails, depending on whether or not text is a sentence in the goal grammar.

*Comments:* 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. Write a main procedure that accepts specifications for goals and text and calls recogn accordingly.

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

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

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

## Problem 10: 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 20, which should be consulted for a more detailed description.

Grammatical specifications are read in and analyzed. A stack of alternatives is created for each definition. Each alternative, in turn, is represented by a stack 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 10 sentences from the language defined by <s>.

The generation process consists of selecting an alternative for the nonterminal and stacking the corresponding subsequents. The stack is then popped. If the popped value is a terminal, it contributes to the evolving sentence. If the popped value is a nonterminal, an alternative is selected for it, its subsequents are pushed, and so on.

```
global def

record nonterm(ntname)

procedure main()
   def := table()
   repeat {
      writes("*")
      line := read() | break
      enter(line) | generate(line) | write("*** syntax error")
      }
end

procedure enter(s)
   local name
   scan s using {
      ="<" | fail
      name := tab(find(">::=")) | fail
      move(4)
      def[name] := buildalt(tab(0))
      }
end
```

```
procedure buildalt(s)
   local k
   k := stack()
   every push(k,buildsub(genalt(s)))
   return k
end

procedure buildsub(s)
   local k
   k := stack()
   every push(k,gensub(s))
   return k
end

procedure genalt(s)
   local t
   scan s || "|" using
      repeat {
         t := tab(find("|")) | break
         suspend t
         move(1)
         }
   fail
end

procedure gensub(s)
   local t
   scan s || "<" using
      repeat {
         if ="<" then {
            t := nonterm(tab(find(">"))) | break
            move(1)
            }
         else t := tab(find("<"))
         suspend t
         }
   fail
end

procedure generate(s)
   local name, count
   scan s using {
      ="<" | fail
      name := tab(find(">")) | fail
      move(1)
      count := integer(tab(0)) | fail
      }
   every 1 to count do write(synthesize(name))
end
```

```
procedure synthesize(s)
   local sentence, nexta, t, y, x
   nexta := stack()
   push(nexta,nonterm(s))
   while t := pop(nexta) do
      if type(t) == "nonterm" then {
         x := def[t.ntname]
         if null(x) then {
            write("*** <",t.ntname,"> undefined")
            fail
            }
         y := x[random(size(x))]
         every push(nexta,!y)
         }
      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.

Open lists could be used in place of stacks for representing the structure of the nonterminals, but this complicates the code somewhat.

*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 stack during generation. 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.

# 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 3.1.1, 3.2.1, and 4.2.1 for a description of literals. Record types are context sensitive; see Section 5.4. See Chapter 9 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 *left-bracket*, *right-bracket*, and *period* indicate occurrences of the characters [ , ] , and . , which have metalinguistic uses in the syntax description language.

*program* ::= *declaration* ...

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

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

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

*record* ::= **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* | ( *expr* )

*literal* ::= *integer-literal* | *real-literal* | *string-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* | *scan-using* | *transform-using* | *fails* | *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*

*scan-using* ::= **scan** *expr* **using** *expr*

*transform-using* ::= **transform** *expr* **using** *expr*

*fails* ::= *expr* **fails**

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

*next* ::= **next**

*break* ::= **break**

*return* ::= **return** *optexpr* | **suspend** *optexpr* | **fail**

*compound-expr* ::= { *optexpr* [ ; *optexpr* ] ... }

## Precedence and Associativity

The relative precedence of control structures, operators, and expression-list delemiters 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 | | |
| *scan-using* | 1 | | |
| *return* | 1 | | |
| *suspend* | 1 | | |
| *fail* | 1 | | |
| & | 2 | infix | left |
| := | 3 | infix | right |
| <- | 3 | infix | right |
| :=: | 3 | infix | right |
| <-> | 3 | infix | right |
| +:= | 3 | infix | right |
| -:= | 3 | infix | right |
| *:= | 3 | infix | right |
| /:= | 3 | infix | right |
| %:= | 3 | infix | right |
| ^:= | 3 | infix | right |
| ++:= | 3 | infix | right |
| --:= | 3 | infix | right |
| **:= | 3 | infix | right |
| \|\|:= | 3 | infix | right |
| *to-by* | 4 | | |
| \| | 5 | infix | left |
| = | 6 | infix | left |
| ~= | 6 | infix | left |
| < | 6 | infix | left |
| <= | 6 | infix | left |
| > | 6 | infix | left |
| >= | 6 | infix | left |
| == | 6 | infix | left |
| === | 6 | infix | left |
| ~== | 6 | infix | left |
| ~=== | 6 | infix | left |
| \|\| | 7 | infix | left |
| + | 8 | infix | left |
| - | 8 | infix | left |

| | *precedence* | *type* | *associativity* |
|---|---|---|---|
| ++ | 8 | infix | left |
| | 8 | infix | left |
| * | 9 | infix | left |
| / | 9 | infix | left |
| % | 9 | infix | left |
| ** | 9 | infix | left |
| - | 10 | infix | right |
| *fails* | 11 | | |
| ! | 12 | prefix | |
| - | 12 | prefix | |
| & | 12 | prefix | |
| \| | 12 | prefix | |
| . | 12 | prefix | |
| + | 12 | prefix | |
| - | 12 | prefix | |
| = | 12 | prefix | |
| ( ... ) | 13 | | |
| [ ... ] | 13 | | |
| . | 14 | infix | left |

## Reserved Words

The following reserved words cannot be used as identifiers:

| | | | | |
|---|---|---|---|---|
| **break** | **else** | **if** | **record** | **then** |
| **by** | **end** | **initial** | **repeat** | **to** |
| **case** | **every** | **local** | **return** | **transform** |
| **default** | **fail** | **next** | **scan** | **until** |
| **do** | **fails** | **of** | **static** | **using** |
| **dynamic** | **global** | **procedure** | **suspend** | **while** |

# APPENDIX B

# Built-In Operations

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

## Functions

| *function* | *section* |
|---|---|
| any(c,s,i,j) | 4.7.2 |
| bal(c1,c2,c3,s,i,j) | 4.7.2 |
| center(s1,i,s2) | 4.5.3 |
| close(x) | 5.1.3, 5.2.3, 6.1 |
| copy(x) | 7.4 |
| cset(x) | 4.3 |
| display(i,f) | 8.4 |
| exit(i) | 11.5 |
| find(s1,s2,i,j) | 4.7.1 |
| image(x) | 7.9 |
| insert(s,i) | 4.8.4 |
| integer(x) | 3.4.1 |
| lbound(x) | 5.1.1 |
| left(s1,i,s2) | 4.5.3 |
| lge(s1,s2) | 4.6 |
| lgt(s1,s2) | 4.6 |
| list(i,j) | 5.1.1 |
| lle(s1,s2) | 4.6 |
| llt(s1,s2) | 4.6 |
| many(c,s,i,j) | 4.7.2 |
| map(s1,s2,s3) | 4.5.5 |
| match(s1,s2,i,j) | 4.7.1 |
| move(i) | 4.8.2 |
| null(x) | 7.7 |
| numeric(x) | 3.5 |
| open(x,s) | 5.1.3, 5.2.3, 6.1 |
| pop(k) | 5.3.2 |
| pos(i) | 4.8.1 |
| push(k,x) | 5.3.2 |
| random(i) | 7.5 |
| read(f) | 6.3 |
| reads(f,i) | 6.3 |
| real(x) | 3.4.2 |
| repl(s,i) | 4.5.2 |
| reverse(s) | 4.5.5 |
| right(s1,i,s2) | 4.5.3 |
| size(x) | 4.2.2, 5.1.1, 5.2.2, 5.3.2 |
| sort(x,i) | 5.6 |
| stack(i) | 5.3.1 |
| stop(f,s1,s2,...,sn) | 11.5 |
| string(x) | 4.4.1 |
| system(s) | 7.10 |

| *function* | *section* |
|---|---|
| tab(i) | 4.8.2 |
| table(i) | 5.2.1 |
| top(k) | 5.3.2 |
| trim(s,c) | 4.5.5 |
| type(x) | 7.8 |
| ubound(x) | 5.1.1 |
| upto(c,s,i,j) | 4.7.2 |
| write(f,s1,s2,...,sn) | 6.2 |
| writes(f,s1,s2,...,sn) | 6.2 |

## Infix Operators

| *operator* | *section* |
|---|---|
| := | 2.2.1 |
| <- | 2.9 |
| :=: | 2.2.1 |
| <-> | 2.9 |
| +:= | 7.2 |
| -:= | 7.2 |
| *:= | 7.2 |
| /:= | 7.2 |
| %:= | 7.2 |
| ^:= | 7.2 |
| ++:= | 7.2 |
| --:= | 7.2 |
| **:= | 7.2 |
| \|\|:= | 7.2 |
| \| | 2.6 |
| & | 2.7 |
| + | 3.1.2 |
| - | 3.1.2 |
| * | 3.1.2 |
| / | 3.1.2 |
| ^ | 3.1.2 |
| % | 3.1.2 |
| = | 3.1.3 |
| ~= | 3.1.3 |
| > | 3.1.3 |
| >= | 3.1.3 |
| < | 3.1.3 |
| <= | 3.1.3 |
| ++ | 4.3 |
| -- | 4.3 |
| ** | 4.3 |
| \|\| | 4.5.1 |
| == | 4.6 |
| ~== | 4.6 |
| === | 7.3 |
| ~=== | 7.3 |
| . | 5.4.3 |

## Prefix Operators

| *operator* | *section* |
|---|---|
| + | 3.1.2 |
| - | 3.1.2 |
| \| | 3.1.2 |
| ~ | 4.3 |
| ! | 7.1 |
| = | 4.8.3 |
| & | 2.2.2 |
| . | 2.3.1 |

## Keywords

| *keyword* | *section* |
|---|---|
| &ascii | 4.3 |
| &clock | 7.6 |
| &cset | 4.3 |
| &date | 7.6 |
| &dateline | 7.6 |
| &errout | 6.1 |
| &host | 7.11 |
| &input | 6.1 |
| &lcase | 4.3 |
| &level | 8.3.3 |
| &null | 2.2.2 |
| &output | 6.1 |
| &pos | 4.8.1 |
| &random | 7.5 |
| &subject | 4.8.1 |
| &time | 7.6 |
| &trace | 8.3.4 |
| &ucase | 4.3 |
| &version | 7.11 |

# APPENDIX C

## Summary of Defaults

### Omitted Arguments in Functions

| *abbreviated form* | *equivalent expression* |
|---|---|
| any(c) | any(c,&subject,&pos,0) |
| any(c,s) | any(s,c,1,0) |
| bal(,,,s,i,j)* | bal(&cset,cset("("),cset(")"),s,i,j) |
| bal(c1,c2,c3)* | bal(c1,c2,c3,&subject,&pos,0) |
| center(s,i) | center(s,i,"□") |
| display() | display(1,&errout) |
| find(s) | find(s,&subject,&pos,0) |
| find(s1,s2) | find(s1,s2,1,0) |
| insert(s) | insert(s,&pos) |
| left(s,i) | left(s,i,"□") |
| many(c) | many(c,&subject,&pos,0) |
| many(c,s) | many(c,s,1,0) |
| map(s) | map(s,&ucase,&lcase) |
| match(s1,s2) | match(s1,s2,1,0) |
| match(s) | match(s,&subject,&pos,0) |
| open(s) | open(s,"r") |
| read() | read(&input) |
| right(s,i) | right(s,i,"□") |
| sort(x) | sort(x,1) |
| trim(s) | trim(s,cset("□")) |
| upto(c) | upto(c,&subject,&pos,0) |
| upto(c,s) | upto(c,s,1,0) |

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

*These defaults apply separately and may be used in combination (s, i, and j default to &subject, &pos, and 0 only if all three of s, i, and j are omitted, however). For example, bal() defaults to

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

# APPENDIX D

## Summary of Type Conversions

### Explicit Conversions

There are five explicit type-conversion functions:

```
cset(x)
integer(x)
null(x)
real(x)
string(x)
```

Each of these functions converts csets, integers, ●, real numbers, and strings to the type indicated by the function name. The functions fail for objects of any other type. The success of a conversion operation usually depends on the specific value involved. For example, integer("10") succeeds, but integer("1a") fails. Failure also occurs in numeric conversions if the result would be out of the allowable range for the numeric type.

### Implicit Conversions

Where required by context, implicit conversions are performed automatically for all types corresponding to the type-conversion functions listed above. If such an implicit conversion cannot be made (that is, if the corresponding explicit conversion would fail), an error of the form 10*n* occurs.

# APPENDIX E

# Summary of Error Messages

## Translator Error Messages

The errors messages that may occur during translation follow. Overflow conditions prevent the translator from continuing. In the case of all other errors, translation is continued, but the translated program cannot be used.

end-of-file expected
global, record, or procedure declaration expected
inconsistent redeclaration
invalid argument list
invalid argument or field list
invalid case clause
invalid character
invalid construction
invalid construction in assignment
invalid construction in augmented assignment
invalid construction in by clause
invalid construction in to clause
invalid context for break
invalid context for next
invalid control expression in case expression
invalid control expression in every expression
invalid control expression in if expression
invalid control expression in scan expression
invalid control expression in transform expression
invalid control expression in until expression
invalid control expression in while expression
invalid digit in integer literal
invalid expression list
invalid field list
invalid field name
invalid global declaration
invalid initial expression
invalid integer literal
invalid keyword
invalid keyword construction
invalid local declaration
invalid radix for integer literal
invalid real literal
invalid reference or subscript
invalid repeat expression
invalid subscript
missing argument list in procedure declaration

missing colon in case clause
missing comma or right bracket
missing expression in do clause
missing expression in else clause
missing expression in then clause
missing expression in using clause
missing field list in record declaration
missing left brace in case expression
missing of in case expression
missing procedure name
missing record name
missing right brace in case expression
missing right parenthesis
missing semicolon
missing semicolon or operator
missing semicolon or right brace
missing then in if expression
missing using in scan expression
missing using in transform expression
more than one default clause
out of global symbol table space
out of local symbol table space
out of string space
out of symbol constant table space
out of tree space
unclosed quote
unexpected end-of-file

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

The linker issues one error message, which prevents use of the linked program:

invalid field name

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.

## Program Error Messages

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

### Category 1: Invalid Type or Form

| | |
|---|---|
| 101 | integer expected |
| 102 | real expected |
| 103 | numeric expected |
| 104 | string expected |
| 105 | cset expected |
| 106 | file expected |
| 107 | procedure expected |
| 108 | record expected |
| 109 | stack expected |
| 111 | invalid type to size |
| 112 | invalid type to close |
| 113 | invalid type to open |
| 114 | structure expected |
| 115 | list expected |
| 121 | variable expected |

### Category 2: Invalid Argument or Computation

| | |
|---|---|
| 201 | division by zero |
| 202 | zero second operand to % operator |
| 203 | integer overflow |
| 204 | real overflow, underflow, or division by zero |
| 205 | value out of range |
| 206 | negative first argument to real exponentiation |
| 210 | invalid field name |
| 215 | second and third arguments to map of unequal length |
| 216 | erroneous list bounds |
| 219 | invalid first argument to sort |
| 220 | invalid second argument to sort |
| 221 | invalid second argument to open |
| 222 | invalid argument to system |
| 223 | invalid type for subscripting |
| 231 | by clause equal to zero |

### Category 3: Invalid Structure Operation

| | |
|---|---|
| 301 | table size exceeded |
| 302 | stack size exceeded |

## Category 4: Input/Output Errors

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

## Category 5: Capacity Exceeded

| | |
|---|---|
| 501 | insufficient storage in heap |
| 502 | insufficient storage in string space |
| 503 | insufficient storage for garbage collection |
| 504 | insufficient storage for system stack |

## Acknowledgement

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. The authors are indebted to Madge Griswold for careful readings of drafts.

## 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 SNOBOL3 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. Gimpel, James F. *Algorithms in SNOBOL4*. John Wiley & Sons, New York, New York. 1976. pp. 25-26.

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

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

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

16. Griswold, Ralph E. "Programming Techniques Using Character Sets and Character Set Mappings in Icon", *The Computer Journal*, to appear.

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

18. Korb, John T. *The Design and Implementation of a Goal-Directed Programming Language*. Technical Report TR 79-11, Department of Computer Science, The University of Arizona, Tucson, Arizona. July 1979. pp. 23-27.

19. 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). to appear.

20. Griswold, Ralph E. *String and List Processing in SNOBOL4; Techniques and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1975. pp. 192-200. 5. IBM Corporation. *System/370 Reference Summary*. Form GX20-1850. White Plains, New York. 1976.

# INDEX