The Design and Implementation of a Goal-Directed Programming Language*

John Timothy Korb†

TR 79 - 11

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
June 1, 1979

This report reproduces a dissertation, titled as above, submitted to The University of Arizona in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

*This work was supported by the National Science Foundation under Grant MCS75-01307.

†Author's present address: Xerox Corporation, 3333 Coyote Hill Rd., Palo Alto, CA 94304.

ABSTRACT

Many typical programming problems are quite difficult, if not impossible, to solve from a purely analytical point of view. The most feasible solution may not involve rigorous analysis of the problem in order to find the precise solution in the most general case, but instead may rely on heuristics to develop a solution through trial and error. Problems of an essentially combinatorial nature, such as string scanning, are likely candidates for this approach. Solutions to these problems are often termed goal-directed, since repeated attempts at a solution are made until a goal is reached. This dissertation describes linguistic facilities that form the basis of the goal-directed facilities for the Icon programming language.

In Icon, goal-directed programming is based on generators, expressions that are capable of producing more than one value. Generators form an integral part of the language and can be used anywhere a value is required. If the value produced by a generator does not yield a successful result, the generator is automatically activated for an alternate value.

A novel string scanning facility using generators is described. Although similar in many respects to the facilities provided by SNOBOL4, the Icon facilities are not based on patterns. Instead, the scanning operations are applied directly to the global subject.

The utility of generators is not limited to goal-directed programming. Examples illustrate how generators can be used in constructs not usually considered related to goal-directed programming.

A method for implementing generators, which has been successfully used in Icon, is described. An advantage of the implementation technique is that little overhead is incurred by expressions that do not use the goal-directed facilities.

ACKNOWLEDGMENTS

The research described in this dissertation was supported by the National Science Foundation under Grant MCS75-01307. Most of the programming was done at the University of Arizona Computing Center. This dissertation was formatted and printed using Xerox word processing equipment. The support of these organizations is gratefully acknowledged.

The Icon programming language, which is described in this dissertation, was designed by Ralph E. Griswold, David R. Hanson, and the author. Walter J. Hansen contributed a number of improvements to the design of the language. Other contributions and suggestions were provided by Cary A. Coutant and Stephen B. Wampler. The implementation was done by Walter J. Hansen, David R. Hanson, and the author. The lively discussions with this group of people contributed directly and indirectly to many of the ideas in this work.

The members of my committee, Christopher W. Fraser and David R. Hanson, have contributed substantially to this work through innumerable suggestions and improvements. David R. Hanson has been a continuous source of energy and ideas, from language design and implementation to help with this dissertation.

Most of all, special thanks go to my advisor, Ralph E. Griswold, for providing many of the ideas central to this work. He has spent countless hours reviewing the many drafts of this dissertation. His patience and understanding throughout this time are gratefully appreciated. It is impossible to fully acknowledge the help he has provided, as computer scientist, author, and friend.

CONTENTS

1. INTRODUCTION

Many typical programming problems are quite difficult, if not impossible, to solve from a purely analytical point of view. The most feasible solution may not involve rigorous analysis of the problem in order to find the precise solution in the most general case, but instead may rely on heuristics to develop a solution through trial and error (Wirth 1976). String scanning problems, such as string analysis and text formatting, provide common situations where such goal-directed programming is often the most practical approach. Artificial intelligence applications also use goal-directed programming for pattern-directed searches of data bases and theorem proving (Bobrow and Raphael 1974).

Goal-Directed Programming

In goal-directed programming, the solution to a problem is formulated as a series of successive approximations to a final goal. The desired goal is instantiated and the program provides sequences of attempts at solutions to that goal. In goal-directed programming languages, the evaluation mechanism provides an environment in which goal-directed programming may be easily phrased.

The key feature of goal-directed linguistic facilities is the provision of a mechanism for making tentative decisions. Instead of requiring that an operation return a single value to be used in subsequent computations, the language allows the programmer to specify a tentative value. Evaluation proceeds as in standard programming languages: operands are combined by operators whose results are then used in subsequent computations. Results of expressions are used to control program flow and determine which operations are to be performed next.

Goal-directed programming languages use the tentative values produced to determine whether the desired goal will be reached or not. Tentative values are generated in a specific language construct, called a decision point (Smith and Enea 1973). This is also known as a goal (Hewitt 1971), tag (Prenner, Spitzen, and Wegbreit 1972), or generator. A decision point produces one of a set of values (perhaps a nondeterministic choice) and allows the computation to continue.

In a programming environment with built-in goal-directed evaluation, the evaluation mechanism automatically extracts alternate values from decision points and reevaluates, at least partially, the expression in which those alternate values are used.

Backtracking

The term backtracking is often used as a synonym for goal-directed programming. During goal-directed programming some reversal of the state of computation may be performed when a decision point is activated to produce an alternate value. If nothing else, control reverts to that point. In addition, the values of some, or all, variables may be restored to the values they had at the time the decision point was initially evaluated. "Backtracking" is used in this dissertation to refer to any effects that are reversed during goal-directed evaluation. Activation of a decision point, then, is the simplest form of backtracking. In complete backtracking, all effects subsequent to a decision point are reversed if that decision point is activated for alternatives. Because of the possibility of input

and output and other "non-reversible" operations, the implementation of truly complete backtracking usually is not possible. However, common operations that cause changes to the state of the computation, such as assignments to variables, can be reversed.

Backtracking is often used in combinatorial analysis; for example, in generating permutations, or in tiling problems (Walker 1960; Lehmer 1960; Hall and Knuth 1965). A mathematical formalization of backtracking appears in Golomb and Baumert (1965). Techniques for estimating the execution time of backtrack programs are described in Knuth (1975).

The implementation of a backtracking scheme with reversal of assignments may be very inefficient. There are two common approaches used to restore effects of assignments. One method is to save the values of all variables and structures that may be modified at the time a decision point is encountered. Another method is to note incremental changes to variables and structures subsequent to a decision point. That is, after a decision has been made, changes that occur in variables and structures are recorded, so that those changes may be reversed.

Opinions vary on whether or not reversal of effects is an integral part of goal-directed programming. Smith and Enea (1973, p. 679) contend that "the main purpose of backtracking is to enable a program to try later alternatives in a goal tree as if earlier unsuccessful ones had never been attempted." Complete reversal of effects, however, discards information; information that may be valuable in making subsequent decisions. Sussman and McDermott (1972, pp. 1172-1173) argue that a program using backtracking "will eventually stumble upon the right path, but its organization makes it hard for it to learn something from an attempt that failed and erased all its side effects."

Although complete reversal of effects does not seem necessary, some backtracking, especially under programmer control, is often useful. A very limited form of backtracking is available in the goal-directed facilities that are the subject of this dissertation. Backtracking is provided with certain string scanning operations and when using a form of assignment that is reversible. Reversible assignment is distinguished from regular assignment and hence must be used explicitly.

Related Research

Goal-directed linguistic facilities exist in a number of programming languages in a variety of forms. Languages intended for research in artificial intelligence typically include some form of goal-directed facilities (Bobrow and Raphael 1974). Backtracking is also used for solving problems in string processing (Gimpel 1976, pp. 43-44).

Most artificial intelligence languages deal with lists and are usually based on the syntax of LISP (McCarthy et al. 1965). PLANNER (Hewitt 1971) and PROLOG (Kowalski 1974) are used for research in theorem proving. These languages provide pattern-directed procedure invocation. In PROLOG, for example, the theorem to be proven is used to select the procedures to invoke. Associated with each procedure is a template that defines the effect of the procedure. The theorem is compared to each template, and those procedures that match are selected. If only one exists, it is invoked. If more than one procedure matches, a choice is made and the selected procedure is invoked. If that procedure fails to prove the theorem, the system backtracks and a different procedure is invoked.

CONNIVER (McDermott and Sussman 1972) is based on PLANNER, but gives the programmer greater control over the search strategy.

SAIL (Reiser 1976) is an Algol-based artificial intelligence language that supports backtracking. The backtracking facilities include primitives to REMEMBER, FORGET, and RESTORE variables in different contexts. Like CONNIVER, SAIL requires explicit manipulation of variables by the programmer to effect a backtracking control regime.

Another approach to backtracking is "multiple-tracking" (Irons 1970). In this technique, when a decision point is reached, rather than choose one value, all values are chosen and execution of each branch proceeds (at least conceptually) in parallel. Multiple-track programming is primarily useful when all branches of a decision tree are to be explored, not just until a successful result is found.

In this dissertation, the deoision point in goal-directed evaluation is called a generator. A number of other languages also use the term generator to describe various language features. In some cases, such as Algol 68 (van Wijngaarden et al. 1976) and Cobol, this use is unrelated to the facilities described in this dissertation.

An early language that has generators is IPL-V (Rand Corporation 1961). In IPL-V a generator is a subroutine that calls a processing routine to operate on each object in a data structure.

More recently, Alphard (Wulf 1974; Wulf, London, and Shaw 1976) and CLU (Liskov et al. 1977) have been developed to support data structure abstraction. In these languages generators are used to iterate over the elements of programmer-defined data structures (Atkinson 1975; Shaw, Wulf, and London 1977). Included with the definition of a data abstraction (or "cluster" in CLU) is a procedure for generating the elements of the abstraction. In CLU, this procedure, called a generator, produces its values using the **yield** statement. When another element is required, execution of the generator continues where the **yield** statement left off. In each of these languages, generators are only accessible in a specific context; a particular type of **for** statement. This restricted access and lack of goal-directed evaluation limit the utility of generators in Alphard and CLU.

Problems With Goal-Directed Programming

As stated in the introduction to this chapter, goal-directed programming facilities often allow a problem to be solved in a simple, straightforward manner, without resorting to a complex theoretical analysis. The saving in programmer time, however, is often reflected in an increase in computer time required to solve the problem. For programs with short lifetimes or programs that are not frequently used, this cost can often be ignored. In other cases, a different technique may be required to solve the problem.

Goal-directed programming techniques have been used extensively for problem solving in artificial intelligence. In these applications, goal trees may be large and complex, and backtracking is often a slow, laborious process (for the program, if not the programmer). For this reason, research in the design of artificial intelligence languages has sought alternate means for exploring goal trees. For example, CONNIVER does not use backtracking. Instead, it computes the alternatives to be explored at any one level, and then iterates over those alternatives (Sussman and McDermott 1972).

Backtrack programming techniques appear in top-down parsing (Gries 1971, pp. 85-92; Aho and Ullman 1977, pp. 174-179). Tentative decisions are made concerning the nature of the input string and actions taken based on those decisions. If a decision turns out to be incorrect, backtracking occurs to the most recent decision point, an alternate decision is made, and parsing continues. In a one-pass compiler, generation of object code is begun

before parsing is completed. If a parsing error is made and backtracking is required, it is awkward, if not impossible, to "back up" the code generator. Backtracking during top-down parsing can often be avoided by rewriting the grammar to avoid redundancies and using recursive-descent parsing (Gries 1971, pp. 93-100; Aho and Ullman 1977, pp. 180-182) or by using "automatic methods", such as LR (Aho and Johnson 1974; Aho and Ullman 1977, pp. 198-245) or SLR (DeRemer 1971) parsers.

Motivation for This Research

Despite these problems, goal-directed programming (and backtracking) remain viable programming techniques. SNOBOL4 (Griswold, Poage, and Polonsky 1971) and, more recently, SL5 (Griswold 1976b, Hanson and Griswold 1978; Griswold, Hanson, and Korb 1977; Hanson 1976a, 1976b) are two languages that support goal-directed programming for string processing. Such languages fill a need for goal-directed programming in areas other than that of artificial intelligence applications or programming language parsers.

Still, SNOBOL4 and SL5 have deficiencies that inhibit their use for goal-directed programming. Both languages suffer from a lack of uniformity in their handling of goal-directed facilities. Their string scanning facilities form a distinct sub-language and are not well integrated into the languages (Griswold 1978).

The desire for a language that would be an effective tool for research in goal-directed programming methods motivated the development of a new programing language, Icon. In Icon generators may appear anywhere in a program. Their use is not restricted to pattern matching, as in SNOBOL4, nor are they as cumbersome to use as coroutines in SL5.

Organization of This Dissertation

The next two chapters describe the Icon language. Chapter 2 gives an overview of the basic language features. Chapter 3 describes in detail the operation of generators. The use of generators in string scanning is described in Chapter 4. Chapter 5 suggests some speculative features for goal-directed programming. Chapter 6 describes the implementation of Icon in general and of generators in particular. Conclusions and final remarks appear in Chapter 7.

2. INTRODUCTION TO ICON

This chapter presents an overview of the Icon programming language. Generally, this presentation is informal, summarizing the salient features and ideas in the language. Specific details are avoided when they are not required for the exposition of generators in the following chapters. The Appendix contains an informal, BNF-like description of the language, along with a partial list of built-in functions and operators. An overview of Icon is presented in Griswold, Hanson, and Korb (1979). Complete details are available in the Icon reference manual (Griswold and Hanson 1979).

Icon has its roots in SNOBOL4 and SL5, and is, in part, the result of experience with these languages. It shares many of the philosophical bases of these languages: concise, expressive features, runtime flexibility, support of untyped identifiers and heterogenous structures, and automatic type conversions. The type of a variable is not bound at compile time. A variable may contain a string at one point during execution and an integer at another. Type checking is performed at runtime, with automatic coercion among types provided when appropriate. Icon also contains some data structuring facilities, such as records, stacks, and tables (in the style of SNOBOL4).

Icon has a high-level syntax similar to that of Algol 68 (van Wijngaarden et al. 1976) and Pascal (Wirth 1971). Typical control structures include **if-then-else** and **while**.

Syntax

An Icon program is a sequence of procedure and record declarations. A procedure is an optional set of type and scope declarations followed by a list of expressions. For example

```
procedure lookup(s, t)
   .
   .
   .
end
```

declares lookup to be a procedure with two arguments, s and t.

Execution begins by invoking the Icon procedure "main". Every program must contain such a procedure.

Expressions

Expressions are constructed in the usual way from prefix, infix, and suffix operators, along with procedure invocations, structure references, and reserved-word control structures. All control structures return a value and may be used as expressions. Braces may be used to group sequences of expressions separated by semicolons into a single expression. The value of such a sequence is the value of the last expression.

Keywords

Keywords provide access to environmental and global state information. Keywords are represented by an ampersand followed by a special identifier. Examples are &lcase and &ucase, strings of the lower- and upper-case letters, respectively; &trace, a variable that controls tracing of user-defined procedures; and &null, whose value is the object of type null. The null object is used as the initial value of all variables and is the identity for many operations (such as addition and concatenation).

Character Strings

Strings are an important component of Icon and many operations deal with strings. A string is a variable-length sequence of characters; the maximum length is limited by available memory.

Substrings and characters in a string are referenced by position specifications. The positions in a string are numbered from the left starting with 1. Positions are between characters; position 1 is before the first character and n + 1 is after the nth character. For example, positions in the string "ABCDE" are numbered as follows.

```
   A   B   C   D   E
   |   |   |   |   |   |
   1   2   3   4   5   6
```

In addition, non-positive integers are used to reference character positions from the right end of the string. Position 0 is the position just past the rightmost character. The string "ABCDE" is equivalently numbered as follows.

```
   A   B   C   D   E
   |   |   |   |   |   |
  -5  -4  -3  -2  -1   0
```

Numbering the positions between characters in a string makes extraction of substrings clear and unambiguous. The built-in function section(s,i,j) returns the section of string s between positions i and j. The function substr(s,i,l) returns the substring of s starting at position i of length l. Thus, substr(s,i,1) returns the ith character of string s.

Character Sets

Character sets are unordered collections of characters and are used in many string operations where the existence of a particular character is important. For example, BREAK(S) in SNOBOL4 depends only on the characters in S, not on their order. The built-in function cset(s) converts the string s into a character set.

Character sets serve two purposes: to allow efficient implementation of certain string analysis functions and to provide a programming tool for a limited form of set manipulation. There are built-in operations for character set union, intersection, and difference.

Conditional Evaluation

As in SNOBOL4 and SL5, Icon uses signals to indicate the success or failure of operations. A language operation is either conditional or unconditional. An unconditional operation, such as addition

```
x + y
```

always succeeds. A conditional operation, such as comparison for equality between numbers

```
i = j
```

may fail. In this case, if the value of i is not equal to the value of j, the operation fails. Similarly, the numeric comparison

```
x < y
```

fails if the value of x is not less than y. The value of a successful comparison is the value of the right operand. Thus, if x is less than y, the expression above evaluates to y. By returning the right operand, comparison operations can be combined, as in

```
a < b < c
```

which succeeds if a is less than b and b is less than c.

Failure of an operation causes control to be transferred to the nearest "expression boundary". Intuitively, "expression boundary" corresponds to the end of the statement in SNOBOL4. For example, in the assignment

```
max := (max < n)
```

if max is not less than n, the operation fails and no assignment is performed. If max is less than n, the comparison succeeds and returns the value of n, which is assigned to max. Thus, the expression above assigns a new value to max only if the new value is larger than the current value. Expression boundaries are described in more detail in the next chapter.

Signal-Driven Control Structures

Like SL5, signals instead of boolean values are used to drive control structures. The **if** expression

if e_1 **then** e_2 **else** e_3

evaluates the expression e_1. If this evaluation succeeds, the expression e_2 is evaluated, otherwise the expression e_3 is evaluated. The result of the **if** expression is the result of e_2 or e_3, whichever is evaluated. The **else** clause may be omitted; it defaults to **else** &null.

The expression

e fails

acts as a signal inverter. If the evaluation of e fails, the expression succeeds and returns the value &null. If the evaluation of e succeeds, the expression fails. This is typically used with the **if** expression

if e_1 **fails then** e_2

which subsumes the more traditional expression

unless e_1 **do** e_2

The **while** expression

while e_1 **do** e_2

evaluates expression e_2 after each successful evaluation of e_1. Using the **fails** operation to invert the signal, the loop

while e_1 **fails do** e_2

subsumes the traditional **until** loop

until e_1 **do** e_2

The **repeat** expression

repeat e

is specifically designed for the signaling mechanism. The expression e is repeatedly evaluated until it fails.

Regardless of the evaluation of their operands, all looping control structures succeed with the value &null.

The two expressions

break
next

are used to modify the iteration of loops. The **break** expression causes an exit from the current (inner-most) loop. The **next** expression transfers control to the end of the current loop, and continues evaluation with the next iteration.

Programming Examples

The following procedures illustrate some of the syntax and control structures described in the preceding sections. The Icon translator allows semicolons to be omitted in cases where the meaning is unambiguous. For this reason, programming examples given here only use semicolons when necessary. See the Appendix for a description of functions and operators.

The first example is a procedure that copies text from file f1 to file f2.

```
procedure copy(f1, f2) local line
  while line := read(f1) do write(f2, line)
end
```

The following version of the copy procedure returns the number of lines copied.

```
procedure copycnt(f1, f2) local line, count
  count := 0
  while line := read(f1) do {
    write(f2, line)
    count := count + 1
    }
  return count
end
```

The procedure copycnt can be improved slightly by combining some operations.

```
procedure copycnt(f1,f2) local count
  count := 0
  while write(f2, read(f1)) do count := count + 1
  return count
end
```

The next procedure takes a non-empty list as argument and returns the average of its elements.

```
procedure average(x)  local sum, i, n
  sum := 0
  i := 1
  while n := x[i] do {
    sum := sum + n
    i := i + 1
    }
  return sum/(i-1)
end
```

Elements of the list are accessed sequentially by incrementing the variable i. When the value of i becomes too large, the access fails, causing termination of the **while** loop. This procedure can be simplified by using the **repeat** control expression, the suffix + operator to increment the variable i, and by relying on the fact that local variables are initialized to &null when the procedure is entered.

```
procedure average(x) local sum, i
  repeat sum := sum + x[i+]
  return sum/(i-1)
end
```

The next example is a filter (Ritchie and Thompson 1974) that reads one file and copies all lines that contain a specified substring to another file. The built-in function find(s_1,s_2) succeeds if string s_1 is a substring of string s_2.

```
procedure filter(f1, f2, s) local line, count
  while line := read(f1) do
    if find(s, line) then {
      write(f2, line)
      count+
      }
  return count
end
```

The value returned by filter is the number of lines actually written to file f2.

A final example is a procedure that implements a simple insertion method to sort a list in ascending order.

```
procedure sort(v) local i, j, key
  j := 1
  while key := v[j+] do {
    i := j
    while key < v[i-] do
      v[i+1] := v[i]
    v[i+1] := key
    }
end
```

The operation of this procedure can be described inductively. Assume the upper portion of the list from v[1] to v[j] is properly sorted. The value of v[j+1] is then inserted in the proper place. The outer **while** loop takes successively larger portions of the list (larger values of j). Each time j is increased, the inner **while** loop finds the correct location for the new value and inserts it there.

3. GENERATORS

A *generator* is an operation that is capable of producing, or generating, a sequence of values. As mentioned earlier, generators form the basis of the goal-directed linguistic facilities in Icon. Typical built-in generators are provided to produce an arithmetic sequence of integers, the positions of particular characters in a string, or the elements of a data structure. These values are produced one at a time as demanded by the expression in which the generator appears.

This chapter describes the semantics of generators; how they interact with each other, with other language operations, and how they are used.

Goal-Directed Evaluation

An expression containing generators represents a goal and the success of such an expression signifies that the goal has been reached. Failure of an expression, on the other hand, implies that the goal has not been reached.

In the absence of failure, operations are evaluated in the standard order; left to right and according to the precedence and associativity of the operators involved. Generators that appear in an expression are evaluated, produce their first value, and become dormant. A *dormant generator* is an operation whose evaluation has produced (at least) one value and has the capacity to produce more, but which is not being presently evaluated.

Failure of some operation initiates goal-directed evaluation in which dormant generators are activated to produce alternate values. The *activation* of a dormant generator transfers control to the point in the expression where that generator became dormant. The generator produces a new value and evaluation continues from there.

The success or failure signal is used to control goal-directed evaluation. As long as operations succeed, evaluation proceeds normally. If an operation fails, however, the activation of dormant generators is begun in order to seek alternatives that may lead to successful evaluation.

Expression Boundaries

As mentioned earlier, failure of an operation generally terminates evaluation of the expression in which it appears. More precisely, failure of an operation causes control to be transferred to the next expression boundary. In SNOBOL4, this corresponds to a transfer to the end of the current statement. In Icon, however, because of the possibility of reactivation of generators, "expression boundary" cannot be defined so simply.

Expression boundaries occur in two places: after expressions in control structures and after expressions separated by semicolons. For example, in the expression below, boundaries are marked by arrows.

```
if a < b < c then { x := a; y := c }
                  ↑        ↑      ↑
```

Similarly, in the three expressions

$$\{\ e_1;\ e_2;\ e_3\ \}$$
$$\uparrow \quad \uparrow \quad \uparrow$$

boundaries appear after each expression. Failure of e_2, for instance, causes control to be transferred to the expression boundary following e_2. Evaluation then continues with expression e3.

Examples of Generators

A simple generator is the alternation

$$e_1 \mid e_2$$

In this expression e_1 is evaluated first. If e_1 succeeds, its value is returned. If e_1 fails, e_2 is evaluated. If e_2 succeeds, its value is returned, otherwise, the expression fails. For example

```
x = 5 | x = 10
```

succeeds if x equals 5 or if x equals 10.

In addition to the usual meaning of alternation described above, the alternation operator is also a generator, generating the values of each of its arguments. If e_1 succeeds, the alternation operator becomes dormant and can be activated to evaluate e_2 if the expression in which it appears fails. This allows the expression above to be rewritten as follows.

```
x = (5 | 10)
```

After generating the first value, 5, the generator becomes dormant while the value 5 is compared to x. If x equals 5, the expression succeeds. If not, the generator is activated and produces the value 10, which is compared to x. If x equals 10 the expression succeeds, otherwise it fails. In either event, evaluation continues with the next expression in sequence.

In order to picture the activity of generators, a notation is used to show active subexpressions and dormant generators. This notation describes the actions of generators by tracing their evaluation and activation. Each step in the evaluation of the expression is shown in a "frame". A frame gives the expression, with the subexpression currently being evaluated underscored. Any dormant generators in the expression are overscored. To the right of the expression is the result of the underscored subexpression.

For example, assuming x is 4 and y is 3, consider the evaluation of the expression

```
x < ((1 | 2) + y)
```

In this example, as in many others involving the alternation generator, the expression 1 | 2 is usually read "one then two", rather than the more conventional "one or two."

1. The generator 1 | 2 is the first operation to be evaluated.

expression	result
x < ((1 | 2) + y)	1

2. The value 1 is produced and the generator becomes dormant. This result is added to the value of y.

expression	result
x < ((1 | 2) + y)	4

3. Next, the value of x is compared to this result.

expression	result
x < ((1 | 2) + y)	failure

4. The comparison fails and the dormant generator 1 | 2 is activated.

expression	result
x < ((1 | 2) + y)	2

5. The generator produces its second value, 2, and has no more values to generate. The generator does not become dormant; it simply returns its value. This value is added to the value of y.

expression	result
x < ((1 | 2) + y)	5

6. The result, 5, is then compared to the value of x.

expression	result
x < ((1 | 2) + y)	success

7. Since 4 is less than 5, this time the expression succeeds and evaluation terminates.

While alternation provides a convenient way to generate a pair of values, a sequence of values can be produced using the sequence generator

e_1 **to** e_2

This expression generates the arithmetic sequence of integers from e_1 through e_2, inclusive. For example, the expression

```
5 to 10
```

generates the sequence 5, 6, 7, 8, 9, 10. An optional **by** clause may be specified to give an increment (or decrement) other than 1.

```
100 to 1 by -1
```

generates the decreasing sequence of integers from 100 down to 1.

Generators can be used to index through the elements of a data structure. The element generator

```
!e
```

generates the elements of object e. The value of e may be a structure, string, or file. For example, if x and y are lists of numbers, the following expression succeeds if they have any equal values.

```
!x = !y
```

For each element of x, all the elements of y are generated and compared to the generated x element. The elements are generated and compared until either two equal values are found, or until all elements have been generated.

If e is a string (or convertible to string), its characters are generated, one at a time. Thus, the expression

```
!i > 5
```

succeeds if the integer i (after conversion to string) contains a digit greater than 5.

Finally, if e is a file, !e generates the lines in that file. For example, the following expression succeeds if string s appears in file f.

```
find(s, !f)
```

Generators for String Analysis

In addition to the alternation, sequence, and element generators, there are several built-in generators for string analysis. In general, these are functions that perform a simple lexical analysis of a string. As generators, they may produce alternate values. The built-in function find(s_1, s_2) returns the position of the first occurrence of s_1 in s_2, or fails if s_2 does not contain the substring s_1. For example

```
find("ab", "abaabbaaabbbaaaabbbb")
```

returns the value 1 initially. If the generator is activated for alternatives, find(s_1,s_2) generates the next position of s_1 in s_2. Repeated activation of the generator above produces the sequence 1, 4, 9, 16.

Another example is upto(c,s) which returns the position of any character in the character set c that appears in s. For example

```
upto("aeiou", "kaleidoscope")
```

returns the value 2 initially. As a generator, upto(c,s) returns successive positions of characters in c in s. The above example yields the sequence 2, 4, 5, 7, 10, 12.

Generator Interaction

In an expression containing several generators, each is invoked, produces a value, and becomes dormant in the order determined by its position in the expression. Failure of an operation initiates the activation of dormant generators. Dormant generators are activated in a last-in, first-out order; the generator that most recently became dormant is the first one activated to produce an alternate value. If that generator has no alternatives, the next dormant generator is activated. This process continues until some generator produces a successful alternate or until no dormant generators remain in the expression. If a successful alternate is found, evaluation continues from that point. Otherwise evaluation of the expression is terminated and control passes to the next expression boundary.

The interaction of generators is illustrated by the expression

```
x = (1 | 2) + (y | z)
```

which contains two generators, 1 | 2 and *y* | *z*. Assuming the variables *x*, *y*, and *z* have the values 6, 3, and 4, respectively, evaluation proceeds as follows.

1. The first generator produces the value 1 and becomes dormant.

expression	result
x = (1 | 2) + (y | z)	1

2. The second generator produces the value of y and also becomes dormant.

expression	result
x = (1 | 2) + (y | z)	3

3. The expression 1 + y is evaluated and the value is compared to x.

expression	result
x = (1 | 2) + (y | z)	4

4. The value of x is not equal to 4, and the comparison fails.

expression	result
x = (1 | 2) + (y | z)	failure

5. Since the comparison fails, the rightmost dormant generator, y | z, is activated.

expression	result
x = (1 | 2) + (y | z)	4

6. The value of z is produced, and the expression 1 + z is evaluated and compared to x.

expression	result
x = (1 | 2) + (y | z)	5

x = (1 | 2) + (y | z)	failure

7. Since x is 6, the comparison fails and, again, the rightmost dormant generator is activated. The generator y | z has no more alternatives, but the generator 1 | 2 has an alternative, 2, which is generated.

expression	result
x = (1 | 2) + (y | z)	2

8. The next operation is the generator y | z, as before. Since it is being initially evaluated again, it produces its first value, y, and becomes dormant. Note that both alternatives of the first generator, 1 | 2, have been evaluated.

expression	result
x = (1 | 2) + (y | z)	3

9. The value of y, 3, is added to the result of the first generator, 2, and the result, 5, is compared to x.

expression	result
x = (1 | 2) + (y | z)	5
x = (1 | 2) + (y | z)	failure

10. The rightmost dormant generator is again activated. The only dormant generator remaining, y | z, is activated to produce its second value, z.

expression	result
x = (1 | 2) + (y | z)	4

11. All generators in the expression have now been evaluated. The value of z, 4, is added to 2, and the result compared with the value of x.

expression	result
x = (1 | 2) + (y | z)	6
x = (1 | 2) + (y | z)	6

12. Since the value produced, 6, equals the value of x, the comparison succeeds and the entire expression succeeds.

Alternation operations can be combined to produce longer sequences of values. For example

```
5 | 9 | 2 | 4
```

produces the sequence 5, 9, 2, 4. Groups of alternations can be used to produce complex comparisons. For example, the expression

```
(a | b | c) = (w | x | y | z)
```

compares the values of a, b, and c with those of w, x, y, and z. If any of the values on the left is equal to any of the values on the right, the expression succeeds, otherwise it fails.

Controlling Generators

Some languages that provide automatic backtracking facilities have been criticized because the facilities tend to get in the programmer's way, not allowing enough control over the generation of alternatives (Sussman and McDermott 1972). Icon provides language constructs to give the programmer control over the evaluation of generators. These constructs can be used to suppress alternatives of a generator, to require that two or more conditions be mutually satisfied, and to force a generator through all alternatives.

Suppressing Alternatives

An important aspect of controlling a generator is being able to discard remaining alternatives if they are not needed. In Icon, dormant generators are discarded when the boundary of the expression in which they appear is crossed. In this way, generators whose values are no longer relevant are not activated in a futile attempt to make another, unrelated expression succeed. In addition, the space required for information associated with dormant generators is also released.

The following expression sequence provides an example.

```
x := upto("aeiou",s1); y := find("begin",s2)
```

The first expression returns the position of the first vowel in s_1 (if one exists). The second expression finds an occurrence of the string "begin" in string s_2. The two expressions are independent; if the second expression fails, remaining alternatives in the first are not tried. In fact, once evaluation has passed through the boundary between the two expressions, the dormant upto generator is discarded; its alternatives are no longer accessible.

Because control structures form boundaries after their arguments, they can also be used to control the activation of generators. For example, the expression

```
if x := upto("aeiou", s1) then y := find("begin", s2)
```

is similar to the example above, except that the second expression is only evaluated if the first expression succeeds. As in the previous example, the two expressions are isolated from one another and cannot interact.

Mutual Success

Often it is useful to test if two or more conditions are true. This is expressed using the conjunction operation

$$e_1 \ \& \ e_2$$

For example, the expression

$$(x > 5) \ \& \ (y < 2)$$

succeeds if x is greater than 5 and y is less than 2.

In the expression

$$e_1 \ \& \ e_2$$

e1 is evaluated first. If e_1 succeeds, e_2 is evaluated. If e_2 fails and e_1 contains any dormant generators, e_1 is activated. If there are none or if e_1 fails, the expression fails. Control flows between e_1 and e_2, activating generators until both expressions succeed or until e_1 fails.

The following expression is one way to determine whether n is divisible by any of several small primes.

```
(p := 2|3|5|7|11) & (mod(n,p) = 0)
```

If any of the primes divides n, the expression succeeds and p is assigned the divisor, otherwise, the expression fails.

Assuming n has the value 35, evaluation proceeds as follows. To simplify the exposition, the generator 2|3|5|7|11 is treated as a single expression that produces five alternate values, rather than as four binary alternation operations.

1. The generator 2|3|5|7|11 produces its first value, 2, and becomes dormant.

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	2

2. The value 2 is assigned to p and control transfers to the divisibility test.

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	2
(p := 2|3|5|7|11) & (mod(n,p) = 0)	2

3. The remainder of 35 divided by 2 is computed.

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	1

4. The result, 1, is compared to 0.

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	failure

5. The comparison fails, and the generator in the first half of the conjunction is activated. It generates the value 3.

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	3

6. The value is assigned to p, the remainder of 35 divided by 3 is computed, and the result compared to 0.

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	3
(p := 2|3|5|7|11) & (mod(n,p) = 0)	3
(p := 2|3|5|7|11) & (mod(n,p) = 0)	2
(p := 2|3|5|7|11) & (mod(n,p) = 0)	failure

7. Again, the comparison fails and the generator is activated. This time the value produced is 5. The assignment and remainder are recomputed.

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	5
(p := 2|3|5|7|11) & (mod(n,p) = 0)	5
(p := 2|3|5|7|11) & (mod(n,p) = 0)	5
(p := 2|3|5|7|11) & (mod(n,p) = 0)	0

8. Finally, the comparison to 0 succeeds

expression	result
(p := 2|3|5|7|11) & (mod(n,p) = 0)	0

and the expression succeeds. The value of p is 5, the first prime divisor of 35.

This example can also be done without using the conjunction operator by including the assignment to p within the invocation of mod.

```
mod(n, p := 2|3|5|7|11) = 0
```

If the value of p is not needed, the expression can be further simplified to

```
mod(n, 2|3|5|7|11) = 0
```

Another example is the expression

```
(i := 1 to 5) & (j := 1 to 5) & p(i,j)
```

which evaluates procedure p at differing values of i and j as long as p(i,j) fails. The value of j varies most frequently, yielding the sequence of calls

```
p(1,1)
p(1,2)
  .
  .
  .
p(1,5)
p(2,1)
  .
  .
  .
p(5,4)
p(5,5)
```

The expression terminates successfully when p(i,j) succeeds for some i and j. If p(i,j) does not succeed, the expression fails after p has been called with all combinations of i and j.

Forcing Alternatives

The **every** expression is used to force a generator to produce all its alternatives. The expression

every e_1 **do** e_2

evaluates e_1 and repeatedly reactivates it until it is exhausted. After each activation of e_1, the expression e_2 is evaluated. For example, if sum is initialized to zero, the expression

```
every i := !x do sum := sum + i
```

finds the sum of the elements of array x.

The **every** expression, in conjunction with generators, provides a facility for composing conventional as well as unconventional control structures from more basic units. Using the **to** generator the equivalent of the Algol **for** statement can be obtained. For example, the expression

```
every i := 1 to 100 do y[i] := x[i]
```

copies list x into list y.

As another example, if x is a list of 100 elements, the expression

```
every i := (1 to 10) | (91 to 100) do write(x[i])
```

writes the first and last ten elements of x.

The **every** expression is more concise than the standard Algol **for** loop in certain applications. The **for**-loop control variable is often just an artifact that can be omitted. For example, the expression

```
every 1 to 50 do f()
```

invokes the procedure f fifty times.

The **do** clause is optional and can often be omitted, allowing even more concise expressions. For instance, auxiliary variables that are needed in other languages to hold transitory values (such as list indexes) are not always needed in Icon. The example above to write the first and last ten elements of array x can be rewritten as

```
every write(x[(1 to 10) | (91 to 100)])
```

Since the element generator generates the elements of a list as variables, **every** can be used to zero the elements of a list. For example

```
every !x := 0
```

zeros the elements of the list x.

Programmer-Defined Generators

Procedures may be used as programmer-defined generators. In a standard procedure, **return** is used to return a value to the calling procedure. The **succeed** and **fail** expressions are used instead of the **return** expression to explicitly return a success or failure signal. The **return**, **succeed**, and **fail** expressions all cause termination of the procedure in which they appear; the procedure cannot then be used as a generator.

A procedure becomes a dormant generator using the **suspend** expression. The expression

```
suspend e
```

suspends the execution of the current procedure and returns the value of e. The procedure remains dormant until activated for an alternative. If the procedure is activated, the expression e is activated for alternatives. If an alternative of e exists, the procedure suspends again, with that alternate value. If e has no alternatives, control passes to the expression following the suspend expression.

For example, the procedure below generates the (infinite) sequence of Fibonacci numbers.

```
procedure fib
  local u1, u2, f, i
  suspend 1 | 1
  u1 := u2 := 1
  repeat {
    f := u1 + u2
    suspend f
    u1 := u2
    u2 := f
    }
end
```

The first two values of the sequence, 1 and 1, are treated as special cases in the first **suspend** expression. Subsequent values of the sequence are computed and the procedure is suspended in the middle of the **repeat** loop after each computation. The following expression repeatedly assigns values of the Fibonacci sequence to f until a value divisible by n is found.

```
mod(f := fib(), n) = 0
```

Thus, f is assigned the first Fibonacci number divisible by n.

Another example is the procedure getword(f) that generates the words that appear in file f. A word is defined here to be any sequence of upper- and lower-case letters. The version of this procedure that appears below uses the string analysis functions.

This procedure uses a **static** variable initialized by an **initial** expression. Analogous to **own** variables in Algol, **static** variables in Icon are local to the procedure in which they are declared, but retain their values across procedure calls. Variables that are not declared **static** are assumed to be **dynamic** and are initialized to &null each time the procedure is invoked. The **initial** expression

```
initial e
```

may be used to assign values to **static** variables. The first time the procedure in which the **initial** appears is invoked, the expression e is evaluated. On subsequent calls e is no longer accessible. The **initial** expression below initializes wordchar to a character set consisting of the upper- and lower-case letters (the ++ operator performs character set union).

```
procedure getword(f)
  local i, line
  static wordchar
  initial wordchar := &lcase ++ &ucase
  while line := read(f) do {
    i := 1
    while i := upto(wordchar, line, i) do
      suspend section(line, i, many(wordchar, line, i))
    }
  fail
end
```

One use of such a procedure is illustrated by counting the words in a file. If ftab is a table, the expression

```
every ftab[getword(f)] +
```

builds a table containing the count of each word appearing in f.

Generators can also be used to generate elements of a user-defined data structure. Below are two procedures that generate the nodes of a binary tree. Nodes of the tree are represented using records with three fields.

```
record node
  data, ltree, rtree
end
```

The data field contains a string that is the value of the node. The fields ltree and rtree are used as pointers to the left and right subtrees, respectively. A leaf node has ltree and rtree values of &null. The expression

```
null x
```

is used to test for leaf nodes; **null** x succeeds if x has the value &null.

The procedure walk(t), which follows, generates the data fields of all the nodes in tree t. The nodes are generated in postorder, that is, the left and right subtrees of a node are generated, followed by the node itself. The infix dot is used, as in Pascal (Wirth 1971), to access the fields of a record.

```
procedure walk(t)
  if null t.ltree fails then
    suspend walk(t.ltree | t.rtree)
  return t.data
end
```

The **suspend** expression suspends with each of the values generated by walking the left subtree, and then with each of the values generated by walking the right subtree. The **return** expression produces the value of the root node.

The procedure leaves(t) below generates only the leaves of the binary tree t. The leaves are generated as though the tree had been walked in postorder.

```
procedure leaves(t)
  if null t.ltree then
    return t.data
  else {
    suspend leaves(t.ltree | t.rtree)
    fail
    }
end
```

The **suspend** expression repeatedly suspends first with the leaves of the left subtree and then with the leaves of the right subtree. After all leaves are generated the procedure fails, indicating that there are no more values. A use of this procedure appears in Chapter 5.

Backtracking with Generators

The goal-directed evaluation of generators, which extracts alternate values when an expression fails, can also be used to reverse the effects of an operation. A generator to be used for backtracking typically modifies some data structure and becomes dormant. If activated for alternatives, it then reverses the modification and fails. This failure indicates that the generator did not produce an alternate value, and causes the activation of other dormant generators.

An example of a backtracking operation is the reversible assignment

e_1 <- e_2

The value of e_2 is assigned to the variable e_1 and the assignment operator becomes dormant. If activated for alternatives, the previous value of e_1 is restored and the expression fails.

For example, suppose a program is to read a parameter from the input file and assign the value to a global variable. If the line is of zero length, a default value, which was assigned earlier, is to be retained. The expression

```
(param <- read()) & (size(param) ~= 0)
```

assigns the next input line to the variable param. If the line is of zero length, the assignment is reversed and the original (default) value is retained. The use of reversible assignment obviates the need for a temporary variable.

Although Icon provides only limited backtracking, more elaborate backtracking facilities can be constructed using programmer-defined generators. Such a backtracking generator is composed of three parts.

```
procedure ...
        .
        .
    <save data structure>
        .
        .
    <modify data structure>
        .
        .
      suspend
        .
        .
    <restore data structure>
        .
        .
      fail
end
```

The first section performs the data structure modifications required by the problem and suspends execution of the procedure. If the calling expression fails, control returns to the generator for an alternative. If the generator has more alternatives, they are produced. Otherwise, the generator restores the data structure to its original condition and fails. The failure signal of the generator is passed to the calling expression and causes continuation of the search for alternatives through the activation of other generators. Thus, unlike in many other languages, reversal of effects is explicitly controlled by the Icon programmer.

An example in which generators can be used for backtracking is the eight queens problem (Dahl, Dijkstra, and Hoare 1972). The object is to place eight queens on a chess board such that no queen can attack any other.

The standard method of solving this problem uses a recursive procedure to place queens on successive rows, retreating to a previous position if a row becomes blocked. For example, the following procedure is based on the solution given by Wirth (1976).

```
procedure queen(c)
  local r
  every r := 1 to 8 do
    if test(r, c) then {
      occupy(r, c)
      if c = 8 then
        printsolution()
      else
        queen(c + 1)
      release(r, c)
      }
end
```

The procedures test, occupy, and release contain the details of the board representation. Procedure test(r,c) succeeds if a queen can be placed on the given row and column. Procedures occupy(r,c) and release(r,c) modify the data structures associated with the board to indicate the presence or absence of a queen on the selected square. The procedure to print the solution is called when the eighth queen is successfully placed on the board.

Following Wirth (1976), three lists are used to keep track of the free rows, the upward facing diagonals, and the downward facing diagonals. If the square at row r and column c is free, then each of the list elements

```
rows[r]
up[r-c]
down[r + c]
```

is null; otherwise, at least one is non-null. The procedure test(r,c)

```
procedure test(r, c)
  if null(rows[r]) & null(up[r-c]) & null(down[r + c]) then
    succeed
  else
    fail
end
```

succeeds if these conditions are met.

The procedure occupy(r,c) assigns a non-null value (the integer 1) to the three selected lists elements, and release(r,c) assigns a null value to those three elements.

```
procedure occupy(r, c)
  rows[r] := up[r-c] := down[r + c] := 1
end

procedure release(r, c)
  rows[r] := up[r-c] := down[r + c] := &null
end
```

The expression

```
queen(1)
```

computes and writes (using the procedure printsolution) the first solution, the list of row numbers on which to place the queens.

The procedure queen is obscured by the fact that it must not only place a queen, but must also invoke subsequent queens and handle the backtracking. The queens are not independent pieces; they are part of a hierarchy that forces them to be interdependent.

A better solution, using coroutines, is described in Hanson (1976b). The coroutine solution eliminates the awkward hierarchical relationship among the queens and allows the main program to control the backtracking. Each coroutine is still responsible for reversing the effects that it causes, but the main program determines when this is to be done.

The procedure below is similar to the SL5 solution, but is further simplified by the use of generators.

```
procedure q(c)
  local r
  every place(r := 1 to 8, c) do
    suspend r
  fail
end
```

The value returned by q(c) is the row on which queen c was placed. The expression

```
write(q(1), q(2), q(3), q(4), q(5), q(6), q(7), q(8))
```

generates and writes the first solution

```
15863724
```

(multiple arguments to write are concatenated onto the output file).

The procedure place(r,c) controls the test, occupy, and release procedures of the preceeding solution, modifying the board and suspending if a queen can be placed on the specified row and column. If the procedure is subsequently activated, it backtracks, removing the queen from the board, and then fails. Whenever place(r,c) fails, the to generator produces a new row value, and place is called again.

```
procedure place(r, c)
  if test(r, c) then {
    occupy(r, c)
    suspend
    release(r, c)
    }
  fail
end
```

The place procedure modifies the board (using occupy), becomes dormant, and, if reactivated, reverses its modification (using release).

Using the reversible assignment operator, the two procedures occupy and release can be merged directly into the place procedure.

```
procedure place(r,c)
  if test(r,c) then
    every rows[r] <- up[r-c] <- down[r+c] <- 1 do
      suspend
  fail
end
```

The **every** expression initially assigns a non-null value (the integer 1) to each of the three list elements and suspends. If the place procedure is activated, the **every** expression activates the dormant reversible assignment operations, which restore the three list elements to &null. After restoring the variable, the reversible assignment operation fails, so the **do** clause is not evaluated, and the procedure fails.

4. GENERATORS IN STRING SCANNING

String scanning in Icon is related to the pattern matching facilities of SNOBOL4 and the string scanning facilities of SL5, but with significant differences. Pattern matching in SNOBOL4 uses a runtime data representation of string patterns (Griswold 1972; Gimpel 1973). These patterns are interpreted during execution and perform an analysis of the subject string. SL5 uses the coroutine model of string scanning developed by Druseikis and Doyle (1974) and described further in Doyle (1975). In this approach, greater flexibility is achieved by replacing the pattern data objects by coroutine environments (Griswold 1976a, 1976c; Garnaat et al. 1978).

Like SNOBOL4 and SL5, string scanning in Icon is performed on a global subject string. This subject and a global position are referenced implicitly by string scanning operations and serve to obviate the bookkeeping that otherwise would be necessary.

In a radical departure from its predecessors, Icon does not use the concept of "pattern". Instead of constructing a data object (either static as in SNOBOL4 or procedural as in SL5), Icon string scanning operations are applied directly to the global subject (Griswold 1978).

Generators and goal-directed evaluation in Icon replace the search and backtracking of pattern matching in SNOBOL4. This approach has several advantages:

1. String scanning operations may be freely mixed with other language operations, allowing many operators to be eliminated. For example, SNOBOL4 pattern matching has three forms of assignment: P . V, P $ V, and @N. In Icon, the need for many of these operations is removed by integrating the scanning operations with the remainder of the language.

2. The programmer has greater control over the search and backtrack algorithm. Although more detail typically must be given, Icon provides more direct specification of when backtracking is to occur.

3. Since there are no patterns, Icon avoids the inherent inefficiency of constructing patterns at runtime.

4. Programmer-defined procedures may be conveniently used to augment the built-in string scanning repertoire. Procedures allow parameters and local variables in string scanning, neither of which are available in pattern matching.

A more detailed discussion of the pros and cons of patterns as data objects is provided in Griswold (1978).

The Scanning Control Structure

Scanning is invoked by the expression

scan e_1 **using** e_2

This construction establishes the string e_1 as the subject of subsequent string scanning operations, sets the position to 1 (the beginning of the subject), and evaluates the

expression e_2. The value of the scan expression is the value of e_2; if either e_1 or e_2 fails, the scan expression fails. Typically, e_2 contains string scanning operations, but it may be an arbitrary expression containing any language operation.

The subject and position are available globally through the keywords &subject and &pos, respectively. Assignment to &subject automatically sets &pos to 1. Assignments to &pos fail if the assigned position does not lie within the subject.

String scanning operations that appear in the expression e_2 may change the position or search for character sets or substrings in the subject.

String Scanning Operations

One simple string scanning operation is the function match(s), which succeeds if the string s appears in the subject starting at the current position. Thus

```
scan s using match("The")
```

succeeds if s begins with the string "The". If match succeeds, it returns the position in &subject following s.

Many of the built-in string analysis operations can be used as string scanning operations. The string analysis operations typically have arguments that give a string to be analyzed, beginning at a specified position. If these arguments are omitted, the string &subject is used, beginning at &pos.

For example, the function find(s_1,s_2,i) returns the position of the first occurrence of string s_1 in string s_2 after position i. If s_2 and i are omitted, the default is the section of &subject starting at &pos. That is, find(s_1) is equivalent to find(s_1,&subject,&pos). Thus, scanning operations can be expressed concisely by implicit reference to the subject.

Another example is the function upto(c,s,i). If s and i are omitted, upto(c) returns the position of a character in c that appears in &subject following &pos.

Cursor Positioning Operations

The string scanning operations described above do not change the value of &pos; they simply return the value of a specific position. One simple way to change the position is through explicit assignment to &pos. For example

```
&pos := &pos + 2
```

moves the position to the right two places. If the new position does not lie within &subject, the assignment fails and &pos is not changed. In addition to explicit assignment to &pos, there are two functions that change the position in the subject string: tab(i) and move(i).

The function tab(i) sets &pos to i. (If i does not lie within the subject, tab fails and does not change the value of &pos.) Since non-positive integers specify character positions from the right end of the string, tab(0) sets the position to the right end of the subject string. In the expression

```
tab(upto(c))
```

upto(c) returns the position of the next character of the subject that appears in c, and tab sets &pos to that value. This is analogous to the pattern BREAKX(c) in SPITBOL (Dewar 1967).

The function move(i) adds i to the value of &pos. If i is negative, the position is moved to the left. (If the new position does not lie within the subject, move fails and does not change the value of &pos.) For example, the expression

```
move(5) & match(s)
```

succeeds if string s appears 5 positions after the current cursor position.

Both tab and move return as value the section of the subject string between the initial and final values of &pos. These functions can be used to synthesize a string from the subject string.

For example, the expression

```
scan s using
  repeat write("(", move(1), ")")
```

writes each character of s with surrounding parenthesis on separate lines.

As another example, consider the problem of reversing a list of items separated by commas. For example, reversing the string

```
"elide,clone,banyan,soot"
```

yields the string

```
"soot,banyan,clone,elide"
```

The procedure revlist(s) builds its result by repeatedly concatenating the next item in the list to the beginning of the synthesized string.

```
procedure revlist(s)
  local v
  scan s using {
    while v := "," || tab(upto(",")) || v do move(1)
    v := tab(0) || v
    }
  return v
end
```

The while loop is broken when all items up to the last comma have been processed. The final item is then concatenated onto the beginning of the result using tab(0).

String scanning provides a convenient way to implement the procedure getword(f) described in the last chapter. The getword procedure given below uses string scanning to break words out of the input file f.

```
procedure getword(f)
  static wordchar
  initial wordchar := &lcase ++ &ucase
  repeat scan read(f) using
    while tab(upto(wordchar)) do {
      word := tab(many(wordchar))
      suspend word
      }
  fail
end
```

After all lines in the file have been read, read(f) fails, breaking the **repeat** loop and causing getword to fail, indicating that the last word has been generated. The use of a global subject and position removes the need for extra arguments to the upto and many functions. The preceding version requires an auxiliary variable, i, to keep track of the position of the word in the string being scanned. The function section was also used to avoid operating on portions of the string that have already been analyzed.

String Scanning with Generators

Many of the string analysis functions are generators, and this property carries over when they are used as string scanning operations. For example, find(s_1,s_2) generates the positions of string s_1 in string s_2 and find(s) generates the positions of string s in the subject string. Thus, the expression

```
scan s using mod(find("x"), 2)
```

succeeds if some occurrence of "x" appears in the subject at an even position.

The alternation of patterns in SNOBOL4, as in the expression

```
LEN(10) | LEN(5)
```

is analogous to the Icon expression

```
move(10) | move(5)
```

Both expressions attempt to advance the position by first 10 characters, or, if that fails, by 5 characters. The SNOBOL4 expression, however, constructs a pattern that is subsequently applied during pattern matching. In Icon, no pattern is generated; the position change is attempted at the time the expression is evaluated.

The ability to use generators anywhere in an expression provides a linguistic feature not directly available in SNOBOL4 or SL5. In Icon, the above example can be "compressed" into the more concise form

```
move(10 | 5)
```

The generator 10 | 5 initially produces the value 10.

expression	result
move(10 | 5)	10

If the subject has fewer than 10 characters, or &pos is less than 10 characters from the end, move(10) fails.

expression	result
move(10 | 5)	failure

Failure causes activation of the dormant generator 10 | 5, which produces its second alternative, 5.

expression	result
move(10 | 5)	5

Next, move(5) is evaluated.

expression
move(10 | 5)

If move(5) succeeds, the expression succeeds. If move(5) fails, then, because there are no dormant generators in the expression, the expression fails.

A generalization of the example above is the expression

```
move(25 to 5 by -5)
```

This expression initially attempts to advance the position by 25. If that fails, move(20) is attempted. Repeated failure causes move(15), move(10), and, finally, move(5) to be attempted. If any of these attempts succeeds, the expression succeeds, otherwise the expression fails.

Backtracking During String Scanning

Both tab and move perform reversible assignments to &pos. If the expression in which they appear fails, the value of &pos is restored to the value it had before the assignment was made. For example, the expression

```
tab(upto(c)) & match(s)
```

succeeds if a character in c that appears in the subject is followed by the string s. If there is a character of c in the subject, then upto(c) succeeds and tab changes the value of &pos. If string s does not appear as a substring of the subject at that point, then match fails and backtracking occurs. The value of &pos is restored by tab, and upto is activated to find the next occurrence of c.

Restoration of the value of &pos is a property of the tab and move functions. This is not performed automatically as a general feature of goal-directed evaluation. The functions move and tab are similar to the reversible assignment operator. For example

```
tab(n)
```

has the same effect on the position as

```
&pos <- n
```

No reversal is performed for assignments to &pos that use the standard assignment operator. For example, the expression

```
(&pos := 5) & find(s)
```

first assigns 5 to &pos. If this fails, because &subject is fewer than 5 characters, the expression fails; otherwise, find(s) is called. If the string s does not appear after position 5 in &subject, the expression fails, yet the value of &pos remains 5. Compare this with the expression

```
(&pos <- 5) & find(s)
```

which restores the value of &pos to its previous value if the assignment succeeds but find(s) fails.

String Scanning Examples

One example of string scanning is a procedure that provides an extension to the SNOBOL4 pattern ARB. The string scanning procedure, arb(n), is like ARB, except that it initially increments &pos by n (ARB initially does not change the cursor). When activated for alternatives, arb increments &pos by one, and suspends. Each time arb suspends, it returns the section of the subject that has been "consumed". If the new value of &pos does not lie within the subject, arb resets &pos to its initial value and fails.

```
procedure arb(n)
   suspend tab(&pos + n to size(&subject) + 1)
   fail
end
```

The expression

```
scan s using
   x := arb(1) &
   tab(match(x)) &
   (repeat tab(match(x))) &
   &pos = size(&subject) + 1
```

succeeds if the string s consists of two or more repetitions of a nonnull substring, and assigns to x the value of the repeated substring. For example, if the value of s is "123123123", the expression succeeds, and assigns to x the string "123".

The next procedure is a modification of the procedure getword given earlier, and generates the words in the subject string.

```
procedure word
  local w
  static wordchars
  initial wordchars := &lcase ++ &ucase
  while tab(upto(wordchars)) do {
    w := tab(many(wordchars))
    suspend w
    }
  fail
end
```

A final example uses the word generator to produce a count of each of the words in a file.

```
procedure wordcount(file)
   local ftab, wdata
   ftab := table
   repeat scan read(file) using
     every ftab[word()] +
   every wdata := !sort(ftab) do
     write(wdata[2], "    ", wdata[1])
end
```

The **repeat** loop reads the file, uses the word procedure to generate the words in each line, and increments the corresponding table entry. The next expression outputs each entry of the sorted table. The built-in function sort(t) returns a list of the elements of table t, sorted on the reference field. Each element is itself a list with two elements. The first element is the reference (in this case, the word), and the second element is the value (the count).

5. POSSIBLE LANGUAGE EXTENSIONS

The unconventional control structures of Icon suggest a number of possibilities for extensions to the goal-directed facilities. In addition, the evaluation of some expressions involving generators may be counterintuitive. This chapter surveys some of these problems and describes some unimplemented language extensions. The next chapter discusses possible techniques for the implementation of some of these extensions.

Generators in Iteration

The use of generators for goal-directed evaluation is extended considerably by the **every** expression, which forces all alternatives of an expression to be generated. The repeated activation of dormant generators may, however, lead to unintuitive results in certain instances.

The order of evaluation of operations in an **every** expression determines which operations are reevaluated when the expression is activated. Not all operations that are evaluated before a generator is initially activated are reevaluated when the generator is reactivated. Only operations that were evaluated after the initial evaluation of a generator are reevaluated after the generator is reactivated.

For example, the expression

```
every write(read(f) || !s)
```

prints the value of read(f) with each of the characters of string s appended to the end. Since the evaluation of read(f) precedes the evaluation of !s, read(f) is not reevaluated during each iteration of the loop. In this example, only one line is read from file f.

Evaluation of the expressions

```
x := &null
every x := x || "-" || !"abc"
```

follows the same rules, but the results may not be as obvious. Taken as a standard loop in more conventional languages, the expression above appears to assign to x the string "-a-b-c". In fact, x is assigned the string "-c".

The reason for this result is that the evaluation of the subexpression

```
x || "-"
```

precedes the evaluation of the generator !"abc". The entire **every** expression is evaluated as follows.

1. First, the value of x (initially null) is concatenated with a dash.

expression	result
every x := <u>x || "-"</u> || !"abc"	"-"

2. Second, the generator !"abc" is evaluated and becomes dormant.

expression	result
every x := x || "-" || !"abc"	"a"

3. The results of steps 1 and 2 are then concatenated and assigned to x.

expression	result
every x := x || "-" || !"abc"	"-a"
every x := x || "-" || !"abc"	"-a"

4. Next, the **every** control structure activates the dormant generator !"abc".

expression	result
every x := x || "-" || !"abc"	"b"

5. The string "b" is then concatenated to the result from step 1 above. Note that step 1 is not reevaluated.

expression	result
every x := x || "-" || !"abc"	"-b"
every x := x || "-" || !"abc"	"-b"

6. The string "-b" is assigned to x and the **every** expression again activates the dormant generator !"abc". This time the string "c" is generated and concatenated to the result from step 1. Again, step 1 is not reevaluated.

expression	result
every x := x || "-" || !"abc"	"c"
every x := x || "-" || !"abc"	"-c"

7. The final result, "-c", is assigned to x.

expression	result
every x := x || "-" || !"abc"	"-c"

In most cases, the desired result can be obtained by reorganizing the expression. In the example above, "factoring" the generator to the front of the expression and using the **do** clause provide a simple solution.

```
every t := !"abc" do x := x || "-" || t
```

Coupled Generators

The original motivation for generators was the design of goal-directed linguistic facilities. The activation of dormant generators in a last-in, first-out order, insuring that all possible combinations of generators would be tried, was specifically motivated by this application.

The inclusion of facilities in Icon to force all alternatives of an expression adds a new feature that can no longer be accurately labeled goal-directed evaluation. Activation of dormant generators is performed regardless of the success or failure of the expression in which they appear. This is illustrated by the following example to write all combinations of two characters from strings s1 and s2.

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

For example, the expression

```
every write(!"abc", !"xyz")
```

writes

```
ax
ay
az
bx
by
bz
cx
cy
cz
```

It might be expected, on the other hand, that the two generators would be activated in "parallel", generating corresponding characters from s_1 and s_2 with each activation, producing

```
ax
by
cz
```

Such a "coupled activation" of generators would be useful in some circumstances, and another control structure could be provided to perform the coupling. An expression such as

e_1 **with** e_2

could couple generators e_1 and e_2. If this expression were activated for alternatives, both generators e_1 and e_2 would be activated to produce values. Generator e_1 would be activated first. If e_1 succeeded, it would become dormant and generator e_2 would be activated. If e_2 succeeded, the result of the expression would be the result of e_2. If either generator failed, the expression would fail. Thus, the expression would terminate when either of the generators produced its last value.

As an example, consider the procedure collate(s1, s2)

```
procedure collate(s1, s2)
  local c1, c2, s
  every (c1 := !s1) with (c2 := !s2) do
    s := s || c1 || c2
  return s
end
```

which would interleave the characters in s1 with the characters in s2. For example

```
collate(&ucase, &lcase)
```

would produce the string

```
"AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz"
```

Another example is the "same fringe" problem, due to Hewitt (Hewitt and Patterson 1970). The problem is to determine if two binary trees have the same leaves. One simple solution is to generate all the leaves of the two trees into two lists, and then compare the elements of the lists. This solution is impractical if the two trees are very large, and is especially inefficient if the trees differ in the first few leaves.

The procedure same(t1,t2) given below uses coupled generators to walk the two trees, t1 and t2, in lock step. The procedure leaves(t), defined in Chapter 3, is a generator that would produce each of the leaves of the tree t.

```
procedure same(t1, t2)
  every (x := leaves(t1)) with (y := leaves(t2)) do
    if x ~== y then fail
  succeed
end
```

Capturing Generators

The restriction of the scope of a generator to the expression in which it appears is appropriate to the solution of many goal-directed problems. For example, this restriction allows an efficient implementation of backtracking. In some circumstances, however, it would be useful to activate the same generator in several places, or to be able to activate a generator independently of goal-directed evaluation. A general technique that allows the programmer to explicitly control the activation of dormant generators is described in this section.

The expression

```
capture e
```

would return a data object that allows e to be evaluated at a later time. In addition, if e is a generator, alternate values would be produced in the same way, without regard to the evaluation or activation of other generators. Local identifiers in e would be bound in the context in which the **capture** expression appears. The problems associated with the implementation of this binding technique are discussed in the next chapter.

Evaluation and activation of a captured expression would be done using the **activate** expression

```
activate e
```

where e is a value returned by a capture expression. Note that neither e nor the expression **activate** e are generators. After the captured expression has generated all of its values the **activate** expression would fail.

Using **capture** and **activate**, the collate function can be written in a different way.

```
procedure collate(s1, s2)
  local c1, c2, s
  c1 := capture !s1
  c2 := capture !s2
  repeat s := s || activate c1 || activate c2
  return s
end
```

In this example, the generators !s1 and !s2 are contained by the **capture** expression and are not discarded between iterations of the **repeat** expression. The generators would not interact, but instead would be activated in parallel.

The **capture** and **activate** expressions could be used to implement SL5-style coroutines (Conway 1963; Hanson and Griswold 1978). Coroutines are much like programmer-defined generators; they can suspend their execution and be resumed where they left off. Unlike generators, coroutines need not be resumed in a hierarchical fashion. An **environment** for a coroutine is the data associated with one instance of a coroutine procedure. The environment could then be manipulated like other data objects and resumed to continue invocation of the procdure. In Icon, the expression

```
e := capture f()
```

would correspond to the creation of an environment for the procedure f, which is then assigned to e. The expression

```
activate e
```

would correspond to resuming that environment. The expression e could be activated independently of goal-directed evaluation.

In the next example, the values from the same generator are needed in two places. The **activate** expression, as a coroutine resume, is used in both places to extract values from the captured generator. The procedure getword(f), which is defined in the previous chapter, would be used to generate words from file f. The generator would be captured and used to format the words two per line.

```
g := capture getword(file)
repeat write(activate g, "    ", activate g)
```

A possible method for the implementation of **capture** and **activate** is given in the next chapter.

6. IMPLEMENTATION

As is the typical case, the design of the language features described in the preceding chapters was influenced by the model used for their implementation. Initially, an implementation method was sought that would support the fundamental linguistic goals: generators that could be used as arbitrary expressions, and a goal-directed mode of evaluation that would seek successful results. The development of a consistent, general implementation model helped to define the properties of the language features. In addition, the model has prompted the development of several new facilities not previously conceived.

This chapter describes the implementation of generators in Icon. Enough of the implementation is presented to give a reasonably complete description, but unnecessary details are omitted. The presentation is generally "top-down", with general descriptions followed by specific details. Two readings may be required to absorb all the details of the implementation.

As the Icon system evolved, the terminology used to describe language operations evolved, too. The implementation, however, did not change correspondingly, and still uses much of the old terminology. Identifiers and procedure names used in the implementation reflect this old terminology. In the description that follows, the implementation terminology and the terminology of the preceding chapters are both used. Implementation names, although sometimes confusing as to their origin, provide links between the discussion below and the program listings.

The Components of the Icon System

The Icon system consists of two major parts, a translator and a runtime system. The translator converts an Icon source-language program into a Fortran program. The runtime system contains routines referenced by this program. Running an Icon program consists of translating the Icon source, compiling the resulting program, and loading it with the runtime system. The resulting module is the executable Icon program.

The translator and runtime system are implemented in Ratfor (Kernighan and Plauger 1976), a Fortran preprocessor. Ratfor allows programs to be written in a free format, using typical high-level constructs that are found in most modern programming languages. The preprocessor translates Ratfor programs into equivalent Fortran programs. Thus, Ratfor provides a convenient programming environment, while retaining the portability and quality of generated code that is characteristic of Fortran. Since the preprocessor introduces no machine dependencies, the resulting program is as machine independent as the original Ratfor code.

The Fortran program produced by the translator consists primarily of subroutine calls to the Icon runtime system. These subroutines, along with a large number of utilities, implement the built-in operations in the language. A system stack is central to the runtime system. Operands are fetched from memory and pushed onto the system stack, and operations obtain their operands from the stack and leave their results there.

The basic model for the implementation of generators is based on the two-stack approach commonly used to implement backtracking (Prenner, Spitzen, and Wegbreit 1972). In addition to the standard system stack for maintaining active procedure frames, a second control stack is used to hold the information associated with dormant generators.

Format of the Generated Fortran Code

The Fortran code produced by the Icon translator shares common features with code produced by any mechanical translation system: the code is standard, highly stylized, and often very unsophisticated. The code is easy to generate, but hard to read; it is formatted strictly for the Fortran compiler with no comments or extraneous spacing.

An Icon program is translated into a single Fortran subroutine. The subroutine begins with the standard preamble and common declarations. The labeled common block CMAIN contains all variables referenced by the generated code. The general structure of this subroutine is shown below.

```
      SUBROUTINE ICON
      COMMON /CMAIN/SIGNAL,LABEL,FLABEL
      INTEGER SIGNAL,LABEL,FLABEL

      <declarations of source-program literals>
        .
        .
        .

      <initialization of literals above (using DATA statements)>
        .
        .
        .

      CALL IINIT(200,400,100,4000,200)
      CALL SINIT(S,G,P,I,R,L)

      <code to invoke Icon procedure main>
        .
        .
        .

      <generated code for expressions in the source program>
        .
        .
        .

1     LABEL = FLABEL
2     GOTO (1,2,3,4,5,6,..., m), LABEL
      END
```

Following the preamble is the declaration and initialization (through DATA statements) of literal data that appear in the Icon source program. This data includes all strings, integers, and reals that appear in the program, along with data objects representing user-defined procedures and identifiers.

These declarations are followed by two subroutine calls for initialization. The call to IINIT initializes the runtime storage area. This area is a Fortran array, housing separate regions for integers and strings, as well as a general purpose "heap". The first call defines the boundaries of these regions and initializes them for allocation. The call to SINIT copies the data from the Icon source program, which is declared and initialized above, into the

runtime storage region prepared by IINIT. A detailed description of the Icon storage management system appears in Hanson (1979).

The next section of code invokes the Icon procedure "main", transferring control to the Fortran code corresponding to the user-defined main procedure.

The bulk of the generated Fortran code is the result of translating Icon expressions into the corresponding sequences of Fortran subroutine calls and jumps. The code sequences for several expressions are given in a subsequent section.

Finally, two statements are generated to provide for computed transfer of control to any point in the generated code. The integer m is the maximum label number allocated by the translator. The manner in which these statements are used is described below.

Operation of the Runtime System

The following explanation of the runtime system is mainly concerned with the implementation of generators. Toward that end, however, it is necessary to describe a number of other aspects of the system.

Transfer of Control

Because Icon implements a number of linguistic features that cannot be expressed directly in terms of Fortran language constructs (for example, recursive procedures) various techniques must be used to handle common problems. One problem is the implementation of arbitrary, computed transfer of control to any Fortran statement appearing in the body of the generated code. This transfer is accomplished using the computed goto given in the outline above. Any Fortran statement that is a destination of the computed goto (for example, a procedure entry point) is preceded by a statement of the form

```
n CONTINUE
```

where n is an integer. These labels are sequential integers that are allocated as needed by the translator.

A special computed goto is generated at the end of the Fortran code. This statement contains each of the labels that may be transferred to at runtime.

```
2 GOTO (1,2,3,4,5,6,..., m), LABEL
```

To transfer to an arbitrary label, the integer value of that label is assigned to the variable LABEL, and a goto to label 2 is executed. Control is then transferred to the desired label. The labels 1 and 2 appear in this statement only to simplify the code generation process; the goto does not transfer to these labels.

Handling of Failure

Implementation of the immediate termination of expression evaluation on failure of an operation is a problem not present in many programming languages. In Icon, this problem is handled through the use of a global failure label. The Icon signal is maintained in the Fortran global variable SIGNAL. After each operation that may fail (and only those operations), the translator generates the Fortran statement

```
IF (SIGNAL .EQ. 0) GOTO 1
```

The current failure label is kept in the Fortran global FLABEL. The statement labeled 1 is the assignment

```
1 LABEL = FLABEL
```

followed immediately by the computed goto described above. Thus, a jump to label 1 assigns the current failure point to the variable LABEL and causes control to be transferred to the corresponding statement.

Failure could be handled by explicit transfer to the nearest failure point, but the more general technique above is used because of the additional flexibility it provides. For example, the ability to manipulate and store the failure label would be used in the implementation of the possible language extensions **capture** and **activate**.

Because Icon uses a stack for storage of intermediate results and because failure causes arbitrary termination of evaluation of the current expression, it is necessary to be able to discard partially computed results that are no longer relevant. On entry to any expression that may fail, the current stack height is saved. When evaluation of the current expression is completed, the stack height is reset, insuring that it is correct and does not get out of phase with other operations.

Sample Code Sequences

Operations in the runtime library get their arguments from the main system stack, and leave their results there. Objects on the stack can be either variables or values; the exact representation is given in Hanson (1979).

The subroutines referenced from the generated code are classified into three types: operand, operation, and control.

Operand Subroutines

There are several kinds of operands that are referenced by the code: strings, integers, reals, and global and local identifiers. A particular subroutine call pushes each kind of operand on the stack. For example

```
CALL XGLOBL(n)
```

is used to push a global identifier. (All subroutines that are referenced from the generated code begin with the letter X.) The value n is a unique integer that identifies the global identifier being referenced. The statement

```
CALL XPSTRG(n)
```

pushes the literal string n onto the stack. The other operand subroutines are

```
CALL XPINTG(n)      # push integer n
CALL XPREAL(n)      # push real n
CALL XLOCAL(n)      # push local identifier n
```

Operation Subroutines

There is a Fortran subroutine for each Icon operator. For example, the subroutine XCAT performs string concatenation; it takes the two string arguments from the stack, concatenates them, and leaves the resulting string there. The code corresponding to the Icon expression

```
"suf" || "fix"
```

is (assuming suitable numbers for string literals)

```
CALL XPSTRG(1)
CALL XPSTRG(2)
CALL XCAT
```

Other typical operations and their corresponding Fortran subroutines include

XADD	addition (+)
XSUB	subtraction (-)
XPLUS	convert to numeric (unary +)
XMINUS	negate (unary -)
XASG	assignment (:=)
XNLT	numeric less than (<)

All of the operations above, except XASG and XPLUS, require values of specific types as operands. These operations assume that the operands on the stack are both of the correct type and are dereferenced (values have been extracted from variables). The translator generates, when required, conversion and dereferencing operations. These operations take the top operand on the stack, convert or dereference it as appropriate, and leave the resulting value in its place. The dereferencing subroutine is XDEREF.

Conversion routines have names of the form XCxxxx, where xxxx is the type of value being converted to. These routines give an error message if the value on the stack cannot be converted to the requested type. Typical conversion subroutines are

```
CALL XCINTG         # convert to integer
CALL XCSTRG         # convert to string
CALL XCNUMR         # convert to numeric
```

The last routine, XCNUMR, is used for polymorphous arithmetic operations and converts its operand to either real or integer as appropriate.

An example of the use of these routines is illustrated by the expression

```
x := "the answer is " || (a + 5)
```

Assuming suitable numbers for literals, the generated code for the expression above is

```
CALL XGLOBL(1)      # push x (assumed to be global)
CALL XPSTRG(1)      # push string "the answer is "
CALL XLOCAL(1)      # push a (assumed to be local)
CALL XDEREF         # fetch the value of a
CALL XCNUMR         # convert the value of a to numeric
CALL XPINTG(1)      # push 5
CALL XADD           # add a+5
CALL XCSTRG         # convert a+5 to string
CALL XCAT           # concatenate "the answer is " to a+5
CALL XASG           # assign the string to x
```

Control Subroutines

The third kind of subroutine referenced by the generated code is the control subroutine. Control subroutines are used to implement the various control structures and perform related stack maintenance operations. An example of a simple control subroutine is XPOP, which removes an unneeded value from the top of the stack.

Two subroutines, XMARK and XDRIVE, are used to handle transfer of control on failure, either by transferring to the next expression boundary or by activating dormant generators. The translator generates calls to these two subroutines around any expression that may fail.

```
      CALL XMARK(n)
       .
       .
       .
      <code for expression that may fail>
       .
       .
       .
n     CALL XDRIVE
```

For expressions that cannot fail, the XMARK and XDRIVE pair is not generated. Thus, none of the overhead required for handling failure of conditional expressions is imposed on unconditional expressions.

XMARK saves the heights of the system and control stacks and the current value of the failure label, FLABEL. The argument to XMARK, n, is assigned to FLABEL, which makes the corresponding call to XDRIVE the current failure point. If an operation fails, the Fortran statement

```
IF (SIGNAL .EQ. 0) GOTO 1
```

which is generated after each operation that may fail, causes control to transfer to the XDRIVE call corresponding to the current expression boundary. XDRIVE resets the previous failure label and, since partially computed results may be left on the stack, resets the height to the level when XMARK was called. For example, the code generated for the expression

```
x := (y < z)
```

is:

```
        CALL XMARK(5)                   # establish 5 as failure label
        CALL XGLOBL(1)                  # push x
        CALL XGLOBL(2)                  # push y
        CALL XDEREF                     # dereference y
        CALL XCNUMR                     # convert y to numeric
        CALL XGLOBL(3)                  # push z
        CALL XDEREF                     # dereference z
        CALL XCNUMR                     # convert z to numeric
        CALL XNLT                       # compare y < z
        IF (SIGNAL .EQ. 0) GOTO 1       # jump if failed
        CALL XASG                       # assign x := z
5       CALL XDRIVE                     # adjust stack height
```

A more complex example is provided by the **if** statement

if x < 0 **then** x := -x **else** x := +x

which translates into

```
        CALL XMARK(6)                   # set FLABEL
        CALL XGLOBL(1)                  # push x
        CALL XDEREF                     # dereference x
        CALL XCNUMR                     # convert x to numeric
        CALL XPINTG(1)                  # push 0
        CALL XNLT                       # compare x < 0
        IF (SIGNAL .EQ. 0) GOTO 1       # jump if failed
6       CALL XDRIVE                     # reset FLABEL and stack
        CALL XPOP                       # discard result of x < 0
        IF (SIGNAL .EQ. 0) GOTO 23000   # jump if failed
        CALL XGLOBL(1)                  # push x
        CALL XGLOBL(1)                  # push x
        CALL XDEREF                     # dereference x
        CALL XCNUMR                     # convert x to numeric
        CALL XMINUS                     # negate x
        CALL XASG                       # assign x := -x
        GOTO 23001                      # skip around else clause
23000   SIGNAL = 1                      # reset signal to success
        CALL XGLOBL(1)                  # push x
        CALL XGLOBL(1)                  # push x
        CALL XDEREF                     # dereference x
        CALL XPLUS                      # convert x to numeric
        CALL XASG                       # assign x := +x
23001   CONTINUE                        # if clauses join here
```

To avoid conflict with the computed goto labels, large numbers are used for labels that are referenced only directly by the generated code.

The Implementation of Generators

As mentioned earlier in this chapter, a second stack, the control stack, is used to maintain the data associated with dormant generators. The system stack could be used, but

elaborate threading, as in Bobrow and Wegbreit (1973), would be required because the two stacks do not operate in parallel.

Saving the Execution State

When a generator is initially invoked, it produces its first value and then prepares for the possibility of subsequent activation. In general, this preparation involves the saving of two kinds of information: (1) data related specifically to the generation of alternatives (for example, arguments and local variables) and (2) partially computed results that otherwise might be consumed before the generator is activated.

The first type of information, local data, is clearly necessary, since these values are required to compute subsequent alternatives. For example, the generator

$$e_1 \text{ to } e_2$$

must initially save the values of e_1 and e_2. Then, when the generator is producing the sequence of integers from e_1 to e_2, it must maintain the current generated value.

Saving the second type of information, partially computed results, is more complex, and requires special consideration. An example is the expression

$$x+2 < (1 \text{ to } 5)$$

At the time the to generator is initially evaluated, the stack contains the values x + 2, 1, and 5. The values 1 and 5 are popped off by the generator and the result, 1, is left in their place. The comparison operation pops the two values, x + 2 and 1, off the stack. If the comparison succeeds, the value 1 is pushed back on the stack, the expression succeeds, and control passes to the next expression boundary. If the operation fails, the dormant to generator is activated. The local variables that are required by the generator and the state of the stack when the generator was initially invoked are restored. Restoring the stack to its previous state requires partially computed results to be saved when a generator becomes dormant. In the example above, the only value on the stack that is not local to the generator is the value of x + 2. This value was "consumed" by a subsequent operation, namely the less-than comparison. The call to XMARK preceding the evaluation of an expression containing generators determines the maximum amount of stack that the enclosed expression can consume.

Activation of Generators

The control stack not only contains the local variables and partially computed results described above, but also a label used to return control to the dormant generator. As generators in an expression are evaluated and become dormant, the data and label associated with them are pushed onto the top of the control stack. When activation occurs, the generator on top of the control stack is restored and activated. This generator is the one that became dormant most recently. Using a stack to store generator data determines the last-in, first-out activation of dormant generators.

Generators are activated by a call to XDRIVE. If the signal is failure when this subroutine is called, it invokes the most recently suspended dormant generator. The algorithm for XDRIVE is outlined in the following Ratfor subroutine.

```
subroutine XDRIVE
  if (signal == FAILURE & dormant generators exist) {
    label = label of dormant generator
    restore stack from top of control stack
    }
  else {                    # expression succeeded or has no generators
    label = 0
    pop return value off top of system stack
    reset stack heights
    push return value on top of system stack
    }
  return
end
```

If the expression succeeds, or if it fails and no dormant generators exist, XDRIVE resets the stack height and moves the return value to the top. The variable label is used to control flow after XDRIVE returns. If label is zero, control passes through the if statement and continues with the next expression in sequence. If label is non-zero, control branches to the specified point in the generated code, activating the dormant generator.

An important feature of this implementation is that very little additional overhead is imposed on an expression that does not contain generators, since calls to XMARK or XDRIVE are not generated if the expression cannot fail. If the expression can fail, but does not contain generators, the only overhead is the code required to check and restore the height of the control stack.

Generation of Values

When a generator is invoked, it computes its first value. Any local variables that may be needed if the generator is subsequently activated are pushed onto the system stack. The generator then performs a "save" operation that pushes onto the control stack that portion of the system stack from the top to the point last marked by a call to XMARK. The portion of the stack that can change is saved, as well as the variables local to the generator that may be needed if it is activated later. The temporary values pushed onto the system stack are then popped off and the first computed value of the generator is pushed on. Evaluation then continues from that point.

If a subsequent operation fails, control is transferred to the call to XDRIVE corresponding to the most recent call to XMARK. XDRIVE detects that a generator has become dormant since the last call to XMARK (by noting that the size of the control stack has increased) and activates that generator. This activation is accomplished by copying the data at the top of the control stack to the top of the system stack, restoring the system stack to the point when the generator was last activated, and then jumping to the location in the code where the generator was called.

As an example, the generated code corresponding to the expression

```
x + 2 < (1 to 5)
```

is given below.

```
      CALL XMARK(7)              # set 7 as failure label
      CALL XGLOBL(1)             # push x
      CALL XDEREF                # dereference x
      CALL XCNUMR                # convert x to numeric
      CALL XPINTG(1)             # push 2
      CALL XADD                  # add x + 2
      CALL XPINTG(2)             # push 1
      CALL XPINTG(3)             # push 5
6     CALL XTO(6)                # generate 1 to 5
      IF (SIGNAL .EQ. 0) GOTO 1  # jump if to exhausted
      CALL XNLT                  # compare x + 2 < (1 to 5)
      IF (SIGNAL .EQ. 0) GOTO 1  # jump if compare failed
7     CALL XDRIVE                # activate generator
      IF (LABEL .NE. 0) GOTO 2   # jump to activate generator
```

Calls to generators in the translated code have the following general form.

```
L CALL Xxxxx(L)
```

where xxxx is the name of the operation and L is a computable label. The algorithm a generator uses is given by the program below, written in informal Ratfor.

The global variable signal corresponds to the Icon signal and is used by the generator to determine its current execution phase. If the signal is success, the generator is being activated for the first time. On initial activation, the generator initializes itself and computes its first value. If the signal is failure, the generator restores local variables from the stack and produces an alternate value.

```
subroutine Xxxxx(L)
  if (signal == SUCCESS) {          # initial call
    initialize local variables
    }
  else {                            # activation call
    signal = SUCCESS                # reset signal to success
    fetch saved local variables and restore
    }
  generate next value
  if (generation succeeded) {
    push local variables onto stack
    call save(L)                    # save stack and label
    pop local variables off stack
    push return value onto stack
    }
  else                              # generation failed
    signal = FAILURE
  return
end
```

If the generation of a value succeeds, and alternate values may later be produced, the generator prepares for subsequent activation. Values of local variables are pushed on the system stack and the portion of the stack to the last marked location is saved on the control stack. The local values saved are then popped off and the return value pushed on. If, instead, the generator fails to produce an alternate value, it sets the signal to failure and returns.

Alternation Generator

The alternation e_1 | e_2 requires a different implementation than other generators. Consider, for example, the consequence of treating alternation like a procedure with two arguments, alt(e_1,e_2).

```
      <evaluate e1>
      <evaluate e2>
L     CALL XALT(L)
      IF (LABEL .NE. 0) GOTO 2
```

If e_1 fails, control is passed to the nearest expression boundary, skipping the call to XALT. The expression e_2 would not be evaluated.

Alternation must be given control before its first argument, e_1, is evaluated. The actual code sequence for the expression above is

```
L1    CALL XALT(L1,L2)
      IF (LABEL .NE. 0) GOTO 2
      <evaluate e1>
      GOTO 23000  # e1 succeeded, skip e2
L2    CONTINUE
      <evaluate e2>
23000 CONTINUE
```

The procedure XALT(L1,L2) is quite simple.

```
subroutine XALT(L1,L2)
  if (signal == SUCCESS) {          # initial call
    call save(L1)                   # save our location
    label = 0                       # fall into e1
    }
  else {                            # activation call
    signal = SUCCESS                # prevent further activation
    label = L2                      # label of e2
    }
  return
end
```

The signal is used to determine whether XALT is being evaluated initially, or is being activated for alternatives. A success signal indicates initial evaluation. The location of the call is saved and control "falls through" to expression e_1 as a result of setting label to 0. If the evaluation of e_1 invokes a generator, the control information for that generator is pushed on the control stack after the information for XALT. If failure occurs, dormant generators in e_1 are activated before XALT.

If XALT is invoked when the signal is failure, the signal is reset to success and the variable label is set to the label of e_2. Since label is non-zero when XALT returns, control transfers to e_2.

Repeated Activation of Generators

Generators are only activated by XDRIVE if the expression in which they appear fails. In **every** e, the expression e is repeatedly activated even if it succeeds. The **every** expression is implemented by noting that it is equivalent to an expression of the form

e & (1 = 0)

Since the comparison always fails, e is continually activated for alternatives until it is exhausted.

The **every** expression is implemented by forcing the signal to be failure after e is evaluated. The generated code produced for the **every** expression above can be outlined as follows.

```
    CALL XMARK(L)                 # set L as the failure label
    <evaluate e>
    SIGNAL = 0                    # force activation of generators
L   CALL XDRIVE
    IF (LABEL .NE. 0) GOTO 2      # activate e
```

The actual code produced for the **every** expression is more complex, since loops may be iterated and exited arbitrarily (using the **next** and **break** expressions).

A Suggested Implementation of Captured Expressions

Chapter 5 describes possible extensions to the goal-directed evaluation facilities of Icon, the **capture** and **activate** expressions. This section discusses a possible implementation strategy for these expressions.

After an expression has produced a value, the XDRIVE subroutine discards dormant generators on the control stack by resetting the height to the height when the XMARK subroutine was called. The **capture** expression can be implemented by saving this information rather than allowing XDRIVE to discard it. Since the lifetime of a captured expression is unlimited, and the system stack and control stack must be reset after the captured expression is left, a natural place to store the control stack fragment is in the heap.

The expression

activate e_1

requires e_1 to be a data object of a special type produced by a **capture** expression. This data object would contain a portion of the control stack at the time the captured expression last produced a value. When the **activate** expression is evaluated, it would place the saved data at the top of the control stack and activate the dormant generators. At this point, the **activate** expression would behave like an ordinary expression containing generators. When evaluation of the activated expression is complete, the stack heights would be properly adjusted and the resulting value would be left on the top of the system stack.

The generated code for the **activate** expression would be

```
    <evaluate e1>
    CALL XMARK(L1)              # establish expression boundary
    CALL XACT                   # activate the expression
L1  CALL XDRIVE                 # reset the stack heights
    IF (LABEL .NE. 0) GOTO 2    # jump to activate expression
```

As described earlier, the subroutines XMARK and XDRIVE serve as boundaries for the evaluation of the expression they surround. In this case, the expression would not be physically enclosed, but instead would be logically enclosed by the calls to XMARK and XDRIVE.

The subroutine XACT would push the contents of the block pointed to by e_1 onto the control stack and set the signal to failure. Expression failure would cause XDRIVE to activate dormant generators; the dormant generators just pushed onto the stack by XACT. The variable LABEL would be set to the activation point and control would pass to the captured expression.

If the captured expression contained no dormant generators, the control stack would not be changed. Since the control stack is empty, XDRIVE would set LABEL to 0, and no activation would occur.

The expression

capture e_2

would be translated into the following code.

```
    CALL XPCAP(L1)              # return initial capture data
    GOTO F1                     # skip over the expression
L1  <code for e2>               # code for the captured expression
    CALL XCAP                   # update the capture data
    GOTO 2                      # return to activation point
F1  CONTINUE                    # continue execution after capture
```

The subroutine XPCAP(L1) would create a block in the heap representing a portion of the control stack. The label L1 is used as the activation point of an otherwise null generator. When XACT is evaluated it would push this label on the control stack, causing the first activation by XDRIVE to start at the beginning of expression e_2.

The XCAP subroutine would update the block initially created by XPCAP. The block would be copied into a larger or smaller block as necessary to hold the dormant generators currently on the control stack.

A problem with the implementation outlined above is related to the FUNARG problem in LISP (Moses 1970): local identifiers that appear in the captured expression would be unbound if the procedure returns. For example, the procedure func(s)

```
procedure func(s)
  return capture !s
end
```

would return a captured generator that would produce the characters of string s. Unfortunately, if the expression were activated, the variable s would no longer exist; the binding of s would have been destroyed when func returned.

Since Icon does not support dynamic binding of identifiers, the only variables that could be in the captured expression would be global variables or local variables, either dynamic or static, of the procedure in which the **capture** appears. The solution to this problem would require **capture** to save the current activation record for the procedure, or at least the variables referenced in the captured expression. If a captured expression were activated, dynamic local identifiers would be bound to this saved data, and evaluation would proceed using these bindings. Since the bindings of global identifiers are not affected by procedure activations and returns, global identifiers in a captured expression would be accessed without problem. Static local identifiers are implemented in a manner similar to global identifiers; hence, access to static identifiers is also unchanged.

7. CONCLUSIONS

Generators provide a linguistic facility that allows concise expression of solutions to problems suitable for goal-directed programming. These problems are typically found in string processing, artificial intelligence, and other areas in which combinatorial searching is performed. Experience with Icon has demonstrated its utility in string processing and problems involving simple combinatorial searches. Problems related to artificial intelligence research have not yet been studied.

The most significant contribution of this work is the development of unified linguistic facilities in which goal-directed evaluation is an integral part. In other languages supporting goal-directed programming, for example MLISP2 (Smith and Enea 1973) and ECL (Prenner, Spitzen, and Wegbreit 1972), the facilities are not well integrated into the language. For example, these languages contain special statements for marking decision points and initiating backtracking.

In Icon, generators are expressions and can be used anywhere a value is required. The manipulation and activation of generators are integrated using the control structures and signaling mechanism of the language.

The utility of generators in Icon and their use in string processing is exemplified by the development of a new string scanning technique not based on patterns. One reason languages use patterns for goal-directed programming is because patterns allow the use of a second, sub-language in which combinatorial searching and backtracking can be more easily described. The disadvantage of this approach is that the overall language structure is made considerably more complex by forcing the programmer to actually write in two separate languages (Griswold 1978).

Icon provides an environment in which string scanning facilities based on active evaluation of scanning operations were created. This facility allows the scanning operations to be integrated with the remainder of the language, giving the programmer as much control over scanning operations as has traditionally been provided over algebraic operations.

In addition to built-in generators for string scanning, Icon provides generators for arithmetic sequences and finite sequences of arbitrary values. These additional generators provide facilities for the solution of more general combinatorial problems.

Another contribution of generators stems from their use in constructs not usually considered related to goal-directed programming. For example, the **every** expression forces all alternatives of generators to be evaluated, not just the ones needed to reach a goal. In addition, generators can be used to build more complex control structures from simpler ones (as in the construction of the Algol **for** statement using the **to** generator and the **every** expression).

The extension of procedures to programmer-defined generators allows the programmer to augment the repertoire of built-in generators. Programmer-defined generators can be used, for example, to add new string scanning functions that include generation of alternatives or reversal of effects.

Some of the inherent inefficiencies of goal-directed facilities in other languages are eliminated by avoiding automatic backtracking and by limiting the scope of generators.

Variables are not automatically restored when a dormant generator is activated. Experience has shown that backtracking often is not necessary, that it can be the source of hidden inefficiencies, and that it often hinders the solution of a problem by destroying information.

The scope of generators is limited to the expression in which they appear. Since expressions may be of arbitrary complexity, this places no restrictions on the programmer, yet it allows unwanted alternatives of dormant generators to be explicitly discarded. Limiting the scope of generators not only prevents unwanted activation, but it also permits the space associated with them to be released. Once an expression boundary is crossed, the space on the control stack occupied by dormant generators for that expression is no longer needed.

Another important aspect of the implementation of Icon is the minor impact of generators on the efficiency of other language operations. Very little overhead associated with generators is imposed on expressions that do not contain generators.

Suggestions for Further Research

There are three evident areas for further research: (1) improvement of the implementation, (2) application of the existing facilities to other problem areas, and (3) improvement of the existing linguistic facilities.

The implementation described in this dissertation is essentially the first complete Icon implementation. (An earlier partial version was implemented as a modification of SL5.) This initial implementation was designed for the experimental development of a new programming language. As such, generality and flexibility of the implementation were of considerable importance. A number of operations that could be performed at compile time are performed at runtime instead to avoid preclusion of unanticipated developments in the language.

For example, the current language does not require failure labels to be pushed on the stack by XMARK when an expression is entered. Instead, explicit jumps to the correct failure label could be generated inline, without using the computed goto. However, if failure labels were implemented in this way, the implementation of **capture** and **activate** as described in Chapter 6 would not be possible.

Another example of how the implementation could be improved is the generated code for procedure calls. Although not all procedures are generators, the code produced at the call site always assumes that the procedure may suspend. Since the presence or absence of a **suspend** expression in a procedure may be determined statically at translation time, this fact could be used to produce shorter code sequences for procedures that are not generators.

As mentioned in Chapter 6, the two-stack approach for the implementation of generators could be combined into a single stack (Bobrow and Wegbreit 1973). Combining these stacks, although making the implementation more complex, would allow better management of the space associated with these structures. In the current implementation, the system stack is expanded, if necessary, to the limits of allocated memory (Hanson 1979). The control stack, however, is allocated in the heap. Expanding the control stack would require a second, larger block to be allocated and the control stack to be copied into this new block. Hence, two copies of the control stack would have to exist (momentarily) at the same time, which might require more storage than would be available.

One of the major areas in which goal-directed programming has been used is in artificial intelligence research. As pointed out above, however, little application has been made of the Icon facilities to these problems. Although the availability of a search and backtrack strategy can certainly help the formulation of problem solutions in this area, it is not clear that Icon has the runtime flexibility that is often utilized in LISP- or SNOBOL4-based systems. For example, Icon does not allow runtime creation of procedures. In addition, patterns and pattern matching are used extensively in artificial intelligence. Problems solved in Icon will need to be recast in more procedural terms, rather than relying on the nonprocedural nature of patterns. In some cases, this recasting may be done automatically. For example, it is fairly simple to write a program to convert SNOBOL4-style patterns into Icon procedure definitions.

Icon provides a limited form of backtracking when using the reversible assignment operator, and the move and tab operations of string scanning. The inclusion of these features is a consequence of similar features in SNOBOL4. More experience with Icon is required to determine the actual need for such backtracking operations.

Although a number of built-in string scanning operations are generators, others also might be useful as generators. For example, the function many(c), which spans a stream of characters of c that appear in the subject, might span fewer and fewer characters each time it is activated for alternatives. On the other hand, making generators of too many operations can be inconvenient for the programmer and a source of subtle bugs. In some cases the alternatives provided by a generator are useful, while in other cases generation results in unnecessary combinatorial processing. The function many(c) is not currently a generator because in most situations if the expression involving many does not initially succeed, the expression will continue to fail even when fewer characters are spanned.

For the programmer skilled in writing SNOBOL4 patterns, Icon presents a new set of challenges. The novice Icon programmer tends to use typical SNOBOL4-style coding practices. String analysis functions are combined like patterns, often using only the control facilities of alternation and conjunction. More experience in string scanning is needed to develop new programming techniques that take advantage of the full power of the language.

APPENDIX: ICON LANGUAGE SUMMARY

This appendix contains a summary of the major portion of the Icon syntax and a list of some of the built-in operators and functions. Complete details are available in the Icon reference manual (Griswold and Hanson 1979).

Syntax

The abbreviated syntax of Icon given below is in an informal BNF notation. Brackets denote optional constructs and ellipses following a group denote repetition. In repeated groups, the last occurrence of the delimiter is omitted. Vertical bars separate alternatives. Metasyntactic characters used literally are underscored. The definitions of the syntactic types <prefix-operator>, <suffix-operator>, <infix-operator>, <identifier>, <integer-literal>, <real-literal>, <string-literal>, and <word> appear in the Icon reference manual.

<program> :: = [<declaration> ;]...

<declaration> :: = <record> | <procedure>

<record> :: = **record** <record-name> [<field-name> ,]... **end**

<record-name> :: = <identifier>

<field-name> :: = <identifier>

<procedure> :: = **procedure** <header> <body> **end**

<header> :: = <identifier> <arg-list> <scope-decl> <initial>

<arg-list> :: = [([<identifier> ,]...)]

<scope-decl> :: = [<global-decl> | <local-decl>]...

<global-decl> :: = **global** [<identifier> ,]...

<local-decl> :: = **local** [<retention>] [<identifier> ,]...

<retention> :: = **static** | **dynamic**

<initial> :: = **initial** <expr>

<body> :: = [<expr> ;]...

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

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

<keyword> :: = & <word>

<operation> :: = <prefix-operator> <expr> | <expr> <suffix-operator>
| <expr> <infix-operator> <expr>

<call> :: = <expr> ([<expr> ,]...)

<reference> :: = <expr> [<expr>] | <expr> . <field-name>

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

<structure> :: = <list> | <table> | <record-object>

<list> :: = **list** [[<lower-bound> :] <upper-bound>]

<lower-bound> :: = <expr>

<upper-bound> :: = <expr>

<table> :: = **table** [<size>]

<size> :: = <expr>

<record-object> :: = <record-name> [<expr> ,]...

<control-struct> :: = <if> | <while> | <every> | <repeat> | <fails> | <null> | <to-by> | <next>
| <break> | <return> | <scan>

<if> :: = **if** <expr> **then** <expr> [**else** <expr>]

<while> :: = **while** <expr> **do** <expr>

<every> :: = **every** <expr> [**do** <expr>]

<repeat> :: = **repeat** <expr>

<fails> :: = <expr> **fails**

<null> :: = **null** <expr>

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

<next> :: = **next**

<break> :: = **break**

<return> :: = **return** [<expr>] | **succeed** [<expr>] | **fail** | **suspend** [<expr>]

<scan> :: = **scan** <expr> **using** <expr>

<compound-expr> :: = { [<expr> ;]... }

Built-in Operators

e_1 := e_2 assigns the value e_2 to the variable e_1 and returns the value of e_2.

e_1 <- e_2 assigns the value e_2 to the variable e_1, returning the value of e_2, but reverses the assignment if the expression in which it appears fails.

e_1 :=: e_2 swaps the values of variables e_1 and e_2 and returns the value of e_2.

e_1 | e_2 generates the alternatives e_1 and e_2.

e_1 & e_2 evaluates e_2 if e_1 succeeds and returns the value of e_2.

e_1 + e_2 returns the arithmetic sum of e_1 and e_2.

e_1 - e_2 returns the arithmetic difference of e_1 and e_2.

e_1 * e_2 returns the arithmetic product of e_1 and e_2.

e_1 / e_2 returns the arithmetic quotient of e_1 and e_2.

e_1 = e_2 succeeds if the arithmetic value of e_1 is equal to that of e_2 and returns the value of e_2.

e_1 ~= e_2 succeeds if the arithmetic value of e_1 is not equal to that of e_2 and returns the arithmetic value of e_2.

e_1 > e_2 succeeds if the arithmetic value of e_1 is greater than that of e_2 and returns the arithmetic value of e_2.

e_1 >= e_2 succeeds if the arithmetic value of e_1 is greater than or equal to that of e_2 and returns the arithmetic value of e_2.

e_1 < e_2 succeeds if the arithmetic value of e_1 is less than that of e_2 and returns the arithmetic value of e_2.

e_1 <= e_2 succeeds if the arithmetic value of e_1 is less than or equal to that of e_2 and returns the arithmetic value of e_2.

e_1 || e_2 returns the result of concatenating the string e_2 onto the end of string e_1.

e_1 == e_2 succeeds if e_1 is lexically equal to e_2 and returns the value of e_2

e_1 ~== e_2 succeeds if e_1 is not lexically equal to e_2 and returns the value of e_2.

e_1 ++ e_2 returns the union of character sets e_1 and e_2.

e_1 ** e_2 returns the intersection of character sets e_1 and e_2.

e_1 -- e_2 returns the difference of character sets e_1 and e_2.

+e returns the arithmetic value of e.

-e returns the arithmetic negative of e.

~e returns the character set complement of e.

!e generates the elements of e if it is a structure, or the characters of e if it is a string.

e+ increments the value of the variable e by one and returns that value.

e- decrements the value of the variable e by one and returns that value.

cset(x) returns the result of converting x to a character set.

find(s_1,s_2,i,j) generates the positions of string s_1 in the section of string s_2 between positions i and j.

many(c,s,i,j) returns the position in string s between positions i and j that is after an initial substring of characters in c.

match(s_1,s_2,i,j) succeeds if string s_1 is an initial substring of s_2 between positions i and j and returns the position after the initial substring.

mod(i,j) returns the remainder of dividing i by j.

move(i) adds the value of i to &pos and returns the section of &subject between the initial and final values of &pos. If the new value of &pos does not lie within &subject, move(i) fails.

pos(i,s) returns the positive equivalent of position i in string s. Fails if i does not lie within s.

read(f) returns the next line from file f. Fails if f is at end of file.

reverse(s) returns the string consisting of the characters in s in reverse order.

section(s,i,j) returns the section of string s between positions i and j.

size(x) returns the size of object x. If x is a structure, size(x) is the number of elements in the structure. If x is a string, or convertible to string, size(x) is the length of the string.

sort(x,i) returns a list of the elements of structure x in sorted order. If x is a table, each element of the sorted list is a two-element list giving the subscript and value of the table element. If i is one, the table is sorted by subscript; otherwise, the table is sorted by value.

substr(s,i,j) returns the substring of string s starting at position i and of length j.

tab(i) assigns the value of i to &pos and returns the section of &subject between the initial and final values of &pos. If the value of i does not lie with &subject, tab(i) fails.

upto(c,s,i,j) generates the positions in string s between positions i and j that contain a character in c.

write(f,s_1,s_2,...,s_n) writes the strings s_1, s_2, ..., s_n on a single line to file f.

LIST OF REFERENCES

Aho, Alfred V. and Steven C. Johnson. "LR Parsing", *Computing Surveys*, 6, June 1974, 99-124.

Aho, Alfred V. and Ullman, Jeffrey D. *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977.

Atkinson, Russ. *Toward More General Iteration Methods in CLU*. CLU Design Note 54, Massachusetts Institute of Technology, Project MAC. September 3, 1975.

Bobrow, Daniel G., and Bertram Raphael. "New Programming Languages for Artificial Intelligence Research", *Computing Surveys*, 6, September 1974, 153-174.

Bobrow, Daniel G. and Ben Wegbreit. "A Model and Stack Implementation of Multiple Environments", *Communications of the ACM*, 16, October 1973, 591-603.

Conway, Melvin E. "Design of a Separable Transition-Diagram Compiler", *Communications of the ACM*, 6, July 1963, 396-408.

Dahl, Ole-Jahn, Edsger W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, Academic Press, London, 1972, 72-82.

DeRemer, Franklin L. "Simple LR(k) Grammars", *Communications of the ACM*, 14, July 1971, 453-460.

Dewar, Robert B. K. *SPITBOL Version 2.0*, SNOBOL4 Project Document S4D23, Illinois Institute of Technology, Chicago, February 12, 1967.

Doyle, John N. *A Generalized Facility for the Analysis and Synthesis of Strings, and a Procedure-Based Model of an Implementation*, SNOBOL4 Project Document S4D48, Department of Computer Science, The University of Arizona, Tucson, February 1975.

Druseikis, Frederick C. and John N. Doyle. "A Procedural Approach to Pattern Matching in SNOBOL4", *Proceedings of the ACM Annual Conference*, November 1974, 311-317.

Garnaat, M. J., W. J. Hansen, R. A. Norden, M. D. Parker, K. L. Tulley, R. C. Wirth. *The Design and Implementation of New String Transformation Facilities for SL5*, Technical Report TR 78-1, Department of Computer Science, The University of Arizona, Tucson, January 13, 1978.

Gimpel, James F. "A Theory of Discrete Patterns and Their Implementation in SNOBOL4", *Communications of the ACM*, 16, February 1973, 91-100.

Gimpel, James F. *Algorithms in SNOBOL4*, Wiley and Sons, New York, 1976.

Golomb, Solomon W. and Leonard D. Baumert. "Backtrack Programming", *Journal of the ACM*, 12, October 1965, 516-524.

Gries, David. *Compiler Construction for Digital Computers*, Wiley and Sons, New York, 1971.

Griswold, Ralph E. *The Macro Implementation of SNOBOL4*, W. H. Freeman and Company, San Francisco, 1972.

Griswold, Ralph E. *String Scanning in SL5*, Technical Report S5LD5a, Department of Computer Science, The University of Arizona, Tucson, June 1976a.

Griswold, Ralph E. "The SL5 Programming Language and Its Use For Goal-Directed Programming", *Proceedings of the Fifth Texas Conference on Computer Systems*, The University Texas at Austin, October 1976b.

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

Griswold, Ralph E. *An Alternative to the Concept of "Pattern" in String Processing*, Technical Report TR 78-4, Department of Computer Science, The University of Arizona, Tucson, April 10, 1978.

Griswold, Ralph E. and David R. Hanson. *Reference Manual for the Icon Programming Language*, Technical Report TR 79-1, Department of Computer Science, The University of Arizona, Tucson, January 9, 1979.

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

Griswold, Ralph E., David R. Hanson, and John T. Korb. "The Icon Programming Language; An Overview", *SIGPLAN Notices*, 14, April 1979, 18-31.

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

Hall, Marshall Jr. and Donald E. Knuth. "Combinatorial Analysis and Computers", Herbert Ellsworth Slaught Memorial Papers, 10, Supplement to *American Mathematical Monthly*, 1965, 21-28.

Hanson, David R. *Procedure-Based Linguistic Mechanisms in Programming Languages*, Ph. D. Dissertation, Department of Computer Science, The University of Arizona, Tucson, 1976a.

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

Hanson, David R. *A Portable Storage Management System for the Icon Programming Language*, Technical Report TR 78-16a, Department of Computer Science, The University of Arizona, Tucson, February 1979.

Hanson, David R. and Ralph E. Griswold. "The SL5 Procedure Mechanism", *Communications of the ACM*, 21, May 1978, 392-400.

Hewitt, Carl. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot", *Proceedings of the International Joint Conference on Artificial Intelligence*, 2, 1971, 167-182.

Hewitt, Carl and Michael Patterson. "Comparative Schematology", *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*, June 1970.

Irons, Edgar T. *Multiple-Track Programming*, Research Report 70-1, Department of Computer Science, Yale University, New Haven, Connecticut, October 1970.

Kernighan, Brian W. and P. J. Plauger. *Software Tools*, Addison-Wesley, Reading, Massachusetts, 1976.

Knuth, Donald E. "Estimating the Efficiency of Backtrack Programs", *Mathematics of Computation*, 24, January 1975, 121-136.

Kowalski, Robert. "Predicate Logic as Programming Language", *Proceedings of the International Federation of Information Processing Congress 74*, August 1974, 569-574.

Lehmer, Derrick H. "Teaching Combinatorial Tricks to a Computer", *Proceedings of Symposia in Applied Mathematics 10: Combinatorial Analysis*, American Mathematical Society, Providence, 1960, 179-193.

Liskov, Barbara, Alan Snyder, Russell Atkinson, and Craig Schaffert. "Abstraction Mechanisms in CLU", *Communications of the ACM*, 20, August 1977, 564-576.

McCarthy, John, Paul Abrahams, Daniel Edwards, Timothy Hart, Michael Levin, *LISP 1.5 Programmer's Manual*, second edition, The MIT Press, Cambridge, Massachusetts, February 1965.

McDermott, Drew V. and Gerald J. Sussman. *The CONNIVER Reference Manual*, Artificial Intelligence Laboratory Memo 259, Massachusetts Institute of Technology, 1972.

Moses, J. "The Function of FUNCTION in LISP", *SIGSAM Bulletin*, 4, July 1970, 13-27.

Prenner, Charles J., Jay M. Spitzen, and Ben Wegbreit. "An Implementation of Backtracking for Programming Languages", *SIGPLAN Notices*, 7, November 1972, 36-44.

Rand Corporation, *Information Processing Language-V Manual*, Edited by Allen Newell, Prentice-Hall, Englewood Cliffs, New Jersey, 1961.

Reiser, John F. *SAIL*, Technical Report, Stanford Artificial Intelligence Laboratory, Computer Science Department, August 1976.

Ritchie, Dennis M. and Ken Thompson. "The Unix Time-Sharing System", *Communications of the ACM*, 17, July 1974, 365-375.

Shaw, Mary, William A. Wulf, and Ralph L. London. "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators", *Communications of the ACM*, 20, August 1977, 553-564.

Smith, David C. and Horace J. Enea. "Backtracking in MLISP2", *Proceedings of the International Joint Conference on Artificial Intelligence*, 3, 1973, 677-685.

Sussman, Gerald J. and Drew V. McDermott. "From PLANNER to CONNIVER -- a Genetic Approach", *Proceedings of the Fall Joint Computer Conference*, 41, 1972, 1171-1179.

van Wijngaarden, Adriaan, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisker, "Revised Report on the Algorithmic Language Algol 68", *Acta Informatica*, 5, January 1976, 1-236.

Walker, R. J. "An Enumerative Technique for a Class of Combinatorial Problems", *Proceedings of Symposia in Applied Mathematics 10: Combinatorial Analysis*, American Mathematical Society, Providence, 1960, 91-94.

Wirth, Niklaus. "The Programming Language Pascal", *Acta Informatica*, 1, January 1971, 35-63.

Wirth, Niklaus. *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

Wulf, William A. *Alphard: Toward a Language to Support Structured Programming*, Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, April 1974.

Wulf, William A., Ralph L. London, and Mary Shaw. *Abstraction and Verification in Alphard*, Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, February 1976.