# Control Mechanisms for Generators in Icon*

*Stephen B. Wampler*

TR 81-18

December 1981

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

# Chapter 1

# Introduction

## 1.1 Expressiveness

The development of high-level languages for computer programming shows a consistent trend toward increased *expressiveness.* Expressiveness is a measure of the ease of using a particular language for describing a process from some *problem domain.* Expressiveness is *not* a measure of language generality. The distinction between general-purpose and special-purpose programming languages is one of *applicability.* Applicability is a measure of the number of problem domains within which a language is expressive.

On a more abstract plane, an expressive language permits users to deal with problems at a level close to their own perception of the problem, and to express solutions to problems in concepts meaningful within the problem domain rather than within the implementation domain.

The acceptance by the programming community of a language is dependent upon a number of factors. Certainly the availability of the implementation of the language has a large impact, as does the efficiency of particular implementations. Nevertheless, the success or failure of a language as a useful and viable tool is directly related to the expressiveness it provides within its problem domain. A language that provides a high degree of expressiveness generally becomes available on a host of machines, and efforts are undertaken to provide more efficient implementations. For example, SNOBOL4 (Griswold, Poage, and Polansky 1971) is available on a number of machines. Because of its popularity, there are now several reasonably efficient implementations for SNOBOL4 (Dewar and McCann 1977; Gimpel 1973).

There are several fundamental aspects of expressiveness. Perlis describes three of these as *terseness, flexibility*, and *composability* (Perlis 1977).

*Terseness.* In any expressive language, it should be possible to describe processes succinctly. Ideally, there should be little information required of a programmer beyond the minimum needed to formulate the underlying algorithm. A programmer should not be forced to provide information that is used solely to simplify translation or to increase efficiency. This is not to say a programmer should be restricted from ever supplying such information, since such information may indeed play a useful role. It is merely recognition of the fact that information such as type declarations and the like belong to the implementation domain, not to the problem domain.

*Flexibility.* High-level languages should have a sufficiently rich set of operations and control structures to permit description of a process in a wide variety of forms. Human beings do not think alike, and an expressive language should permit different persons to describe solutions to problems as they view the solutions.

*Composability.* The development of algorithms is an artistic endeavor requiring imagination, foresight, and a clear grasp of the problem and the tools available for solving the problem (Knuth 1974). A language should assist the user rather than be a hindrance. It should help programmers to formulate and express ideas necessary for the implementation of an algorithm.

A truly expressive language is one in which people can directly compose algorithms and processes. Indeed, it might be said that a person could *think* in such a language.

In order to be a useful tool, a language must be understandable. As such, it should supply enough simplicity and consistency of structure for a person to comprehend all aspects of the language without undue effort.

### 1.1.1 Current Trends in Expressiveness

Current high-level language design has developed along two distinct paths. The conventional path places a premium on efficiency. For example, Algol and Pascal have block structure, which permits an efficient stack organization within the runtime environment, as well as having strict type declarations to permit compile-time type checking.

The philosophy of conventional language design is to provide expressive language features in an efficient manner, even at some cost in expressiveness. Most of the conventional languages are written for production environments, where there is necessarily an emphasis upon efficiency.

The second path is that taken in more unconventional languages. In these languages, the emphasis is upon the above aspects of expressiveness, often at the cost of efficiency. Languages such as APL, LISP, SNOBOL4, SETL (Dewar, Grand, Liu, Schonberg, and Schwartz 1979), and applicative languages such as POPLAR (Morris, Schmidt, and Wadler 1980) provide a great deal of terseness and flexibility, but are most often used in environments where execution speed is less important. The fact that all these languages have devotees is an indication of the success of each in providing terseness and flexibility, as well as indication of the importance of expressiveness.

## 1.2 New Language Concepts

Recently, several useful language features have been added to the repertoire available to language designers. These features serve to increase the expressiveness of languages in which they are incorporated. Three features related to the research presented here are *goal-directed evaluation*, *generators*, and *coroutines*.

### 1.2.1 Goal-Directed Evaluation

Many problems are more easily solved using a combinatorial search instead of using a strictly analytic approach. For example, nondeterministic problems are often best solved using combinatorial algorithms (Floyd 1967). The problem solving process in the combinatorial case proceeds through a series of *decision points*. At each decision point, one choice is made from a set of possible choices. The selected choices describe a *path* through the decision points. A path is complete when no more decision points can be added to the path. If the correct choice is made at each decision point, then the correct solution is produced. An incorrect choice at any decision point produces an incorrect solution. The collection of all possible paths through the decision points is termed the *solution space*.

The combinatorial approach examines all possible paths through the solution space until either a correct solution is produced or all paths have been attempted. The intent of goal-directed programming is to provide a means of automatically searching a solution space for a correct solution.

Goal-directed evaluation is most often accomplished through some form of *backtracking* (Lehmer 1957). If a point in the process of searching the solution space is reached where it can be determined that no correct solution is possible by continuing along that path, the evaluation of the program 'backs up' to the previous decision point and selects a new choice. Backtracking provides a depth-first search of the solution space. There are two forms of backtracking found in goal-directed evaluation: *data backtracking* and *control backtracking*.

Data backtracking is based upon the premise that when evaluation selects a choice from some decision point, it must appear as though that decision point is being reached for the first time, and that choice is the first choice being selected, regardless of how many times the program has actually backtracked to that point. Data backtracking requires that all program data must be 'remembered' when a decision point is reached during evaluation, so that it can be reset during backtracking. Typically, data backtracking is implemented either by reversing the evaluation process or by saving the state of the program data at each decision point. The first approach has the advantage of requiring less storage, but requires every operation in the language have an inverse, and may be considerably slower than the second approach.

While data backtracking is a theoretically clean technique with a precise mathematical description, it has several shortcomings. First, data backtracking can be very inefficient. The amount of information that must be restored or recomputed during backtracking can be considerable. Most languages, such as MLISP2 (Smith and Enea 1973), that claim to implement data backtracking do so in impure form, omitting such actions as restoring the contents of data files during the backtracking process. One of the few languages to implement pure data backtracking is SUMMER (Klint 1979). Second, data backtracking prohibits the decision making process from taking advantage of information acquired during the evaluation of unsuccessful paths. This prevents the use of acquired knowledge as an aid in the selection of intelligent choices.

Control backtracking simply returns control to the last decision point and proceeds with a new choice. This approach eliminates the problems associated with data backtracking, but lacks the same cleanliness since side effects of evaluation can affect the decision making process.

Control backtracking can be easily extended to provide data backtracking capability. The artificial intelligence language LPAK allows the programmer to explicitly declare what information is to be restored during backtracking, thereby providing both data and control backtracking (Mylopoulos, Badler, Melli, and Roussopoulos 1973). A different approach is taken in languages such as PLANNER (Hewitt 1970), ECL (Prenner, Sptizen, and Wegbreit 1972), SAIL (Reiser 1976), and Icon (Griswold, Hanson, and Korb 1981). These languages incorporate control backtracking along with primitive operations for restoring data during the backtracking process.

The acquisition of information during control backtracking has been used successfully to speed up goal-directed evaluation (Lindstrom 1976; Montanegro, Giuliano, and Turini 1974). Lindström's technique is to allow decision points to remember partial computations, so that these computations do not need to be redone during backtracking. Montanegro, Giuliano, and Turini permit decision points to communicate with other decision points to help guide the search through the solution space.

### 1.2.2 Generators

A number of languages, including Icon, have extracted the concept of decision points from the process of goal-directed evaluation. In Icon, these decision points are called *generators* (Korb 1979). Generators can be used in conjunction with Icon's control backtracking mechanism to provide goal-directed evaluation.

Generators need not be used only with backtracking schemes. In IPL-V (Rand Corporation 1961), for example, a generator consists of a subroutine that computes a sequence of data values. Each time a value is computed, the subroutine invokes another routine to operate on that value. CONNIVER (McDermott and Sussman 1972) eliminates backtracking by first computing the set of choices of a generator, and then iterating over these choices. In Alphard (Wulf, London, and Shaw, 1976), CLU (Liskov et al. 1977), and SUMMER generators are independent of goal-directed evaluation. These languages restrict the use of generators to a few special control structures.

### 1.2.3 Coroutines

Coroutines add power to programming languages by providing a means for programmers to treat procedures similar to independent communicating processes (Conway 1963). According to Conway, a coroutine treats all other coroutines with which it communicates as subroutines. Thus each coroutine can be treated as the main process in Conway's view. A more general definition of a coroutine is a process for which the values of local variables are retained even when control is not within that process, and in which execution upon entry continues from the point where control last left that process (Marlin 1980).

Coroutines have been incorporated in a number of programming languages, of which Simula (Dahl 1972), SAIL, and several extensions to Pascal (Lemon 1976; Kriz and Sandmayr 1980) are the most widely used. Each of these languages is representative of a more general class of programming languages designed for particular classes of problems: simulation, artificial intelligence, and systems programming. SL5 (Hanson and Griswold 1978) and ACL (Marlin 1980) include general coroutine mechanisms.

## 1.3 Motivation for This Research

Although a number of languages contain features similar to Icon generators, most fail to provide the control mechanisms necessary to fully utilize the capabilities of generators. In early versions of Icon, generator-based control mechanisms were patterned almost exclusively on the control structures in Algol-like languages. Two of the purposes of this research are to examine generators in Icon in order to develop a better understanding of their operation, and to develop additional control mechanisms to extend their expressiveness. Several of these new control mechanisms have been incorporated into Version 4 of Icon (Coutant, Griswold, and Wampler 1981). A notation for describing some aspects of generator evaluation, descriptions of current control mechanisms for generators, and several novel control mechanisms are presented in Chapter 2.

The control mechanisms described in Chapter 2 extend the expressiveness of generators. Nevertheless, the capabilities of generators remain limited by syntactic constraints. An evaluation mechanism similar to that available with coroutines is needed to eliminate these syntactic constraints. Previous research into coroutines has treated coroutine processing as an adjunct to procedure invocation, where either coroutines are a special class of procedures, or vice versa. For example, in both ACL and SL5 both conventional procedures and generators are treated as special classes of a more general coroutine facility. Part of the research presented in Chapter 3 shows that the fundamental operations involved in coroutine processing are more appropriately part of expression evaluation, and that these operations can be treated independently of procedure evaluation. Additionally, Chapter 3 discusses how these operations can be combined with generators to provide increased expressiveness.

One of the problems associated with goal-directed evaluation has been a lack of efficiency in implementation. Chapter 4 uses several models to develop an efficient implementation of goal-directed evaluation. The most efficient model is used to describe the operation of several of the control mechanisms presented in Chapter 2. The operations presented in Chapter 3 are then incorporated into this model.

# Chapter 2

# Expression Evaluation

The evaluation of expressions is a fundamental aspect of any programming language. Nevertheless, the differences in syntax and semantics among programming languages make comparison of expression evaluation mechanisms difficult. This chapter discusses expression evaluation and introduces a notation for the static description of expression evaluation in Icon. This notation gives insight into the relationships between Icon control mechanisms and those found in more conventional languages. In addition, this notation is used to describe the evaluation of various language primitives and control mechanisms of Icon. Finally, a set of primitive control mechanisms from which the other control mechanisms can be formulated is presented.

The control mechanisms presented in this chapter correspond to those found in Version 4 of Icon. No attempt is made to describe all of Icon, the reader is referred to the reference manual for a detailed description (Coutant, Griswold, and Wampler 1981).

## 2.1 Expression Evaluation

Most programming languages contain expressions that are evaluated to produce *results*. A result can be either a *value* or a *variable*. Expression evaluation in a conventional language such as Algol always produces exactly one result. For example, the result of evaluating 1 + 3 is the value 4. Control structures in such languages are driven by the values of their control expressions. In the Algol expression

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

the comparison of x and y produces a Boolean value that is used to determine whether or not the assignment to z is performed.

Expression evaluation need not produce a result. In SNOBOL4 expression evaluation may produce no result at all. For example, the expression

```
EQ(1,0)
```

does not produce a result. The concept of *failure* in SNOBOL4 is equivalent to failure of an expression to produce a result, while *success* corresponds to the production of a result.

SNOBOL4 lacks conventional control structures such as those in Algol, but instead relies upon the presence or absence of results to control a conditional branching mechanism, as in

```
EQ(X,Y)                    :S(LABEL1)F(LABEL2)
```

which causes a branch to LABEL1 if EQ(X,Y) produces a result and a branch to LABEL2 otherwise.

Failure is also used to control completion of the evaluation of enclosing expressions. For example, in the evaluation of

```
Z = EQ(X,Y) 0
```

assignment of 0 to Z is performed only if X and Y are numerically equal.

The *outcome* of expression evaluation in Icon is either a result or failure to produce a result. Expressions are evaluated in Icon with the goal of producing a result as the outcome (Korb 1979). If the outcome of an expression evaluation is failure to produce a result, then the evaluation is said to *fail*, otherwise the evaluation *succeeds*. As in SNOBOL4, Icon control structures are driven by the presence or absence of results. For example

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

assigns 0 to z if the expression x = y succeeds.

While several of the control structures in Icon resemble control structures in Algol, the use of success or failure to drive control structures involves some subtle differences (Griswold 1980). In particular, control structures in Algol are driven by Boolean values, and hence their control clauses must be expressions that produce Boolean results. Thus,

```
x < y
```

produces a Boolean result in Algol. This, in turn, renders expressions such as

```
x < y < z
```

erroneous in Algol.

However, the outcome of an Icon expression is either failure or a computationally useful result. For example, the expression

```
x < y
```

fails if x is not less than y, but produces y otherwise. This makes it possible to write expressions such as

```
if x < y < z then z := x else z := y
```

Failure may be used as it is in SNOBOL4 to abort the evaluation of enclosing expressions. For example, the function read fails when attempting to read past the end of a file. If read fails during evaluation of

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

the function write is not invoked, and the entire expression fails. Hence

```
while write(read())
```

copies the input file to the output file.

Expressions in Icon are capable of producing several results in sequence and are termed *generating expressions* (Griswold, Hanson, and Korb 1981). Context determines whether evaluation of an expression produces more than one result. The expression every *e* is used to produce the entire sequence of results for *e*. For example, the expression 1 to 10 is capable of producing the results 1, 2, ..., 10. Evaluation of

```
every write(1 to 10)
```

writes the results 1, 2, ..., 10. However, in the expression

```
(1 to 10) >= 3
```

only the results 1, 2, and 3 are produced during evaluation of 1 to 10, since the context only requires a result greater than or equal to 3.

There are no constraints on the length of a sequence of results in Icon. The sequence may be empty, as in evaluation of 1 = 0, or it may contain an infinite number of results.

The expression evaluation mechanisms in Algol-like languages and SNOBOL4 are in a sense subsets of the Icon expression evaluation mechanism. Expressions in Algol-like languages correspond to Icon generators that produce exactly one result. Except in pattern-matching, SNOBOL4 expressions correspond to generators producing at most one result. The pattern-matching component of SNOBOL4 constitutes a sublanguage (Griswold and Hanson 1980) in which some patterns act as generators during the pattern-matching process. Patterns capable of producing more than one result include the patterns for ARB, BAL, and the patterns produced by BREAKX(S) (Dewar 1971) and P1 | P2.

## 2.2 Expression Instances

During the evaluation of an expression, there exist portions of the machine state that are meaningful only during that evaluation of the expression. For example, temporary variables created during that evaluation are not relevant except during that evaluation. These portions of the machine state constitute the *environment* in which that expression is evaluated. An *expression instance* consists of an expression and an environment in which an evaluation of that expression occurs, just as a procedure instance consists of a procedure and the environment in which an invocation of a procedure occurs. The execution of a program proceeds through a sequence of expression instances in much the same way that execution proceeds through a sequence of procedure instances.

There are five states of expression instances:

1. When an expression instance is initially formed, but evaluation has yet to take place within the instance, it is *created*. An expression instance for an expression may be created at any time prior to evaluation of that expression. Typically the expression instance is created just prior to the time the evaluation starts.

2. An expression instance is *active* whenever the corresponding expression is in the process of being evaluated.

3. Because expressions contain other expressions, expression instances are often nested during evaluation. Thus an expression instance may become *passive* while awaiting the result of another instance. For example, assuming evaluation of j + k occurs within a one expression instance, an expression instance for the evaluation of i > (j + k) becomes passive while the instance for (j + k) is active.

4. When an expression instance computes a result but is capable of producing subsequent results, it becomes *inactive*. An inactive expression instance may be *reactivated* to produce a subsequent result. Evaluation within an inactive expression instance is *suspended*.

5. An expression instance that has produced all possible results is *exhausted*. Typically expression instances are destroyed as soon as they become exhausted. An expression instance *exists* between the time it is created and the time it is destroyed.

In Algol-like languages, expression instances become active as soon as they are created, become exhausted as soon as a single result is produced, and are destroyed immediately upon becoming exhausted. Thus all expression instances are either active or passive in Algol-like languages. In Icon, however, expression instances can exist in any of the five states.

Not all expressions require separate expression instances for evaluation, and it is possible to *coalesce* the expression instance for such an expression with any surrounding expression instance. The term *subexpression* is used to refer to an expression that has had its expression instance coalesced with that of some enclosing instance. For example, evaluation of

```
x < y < z
```

occurs within a single expression instance and consists of two subexpressions: one comparing x and y and the other comparing the result of the first subexpression with z.

Expression instances in Icon can be coalesced up to the points at which program control decisions are made. Typical control decision points are the control clauses of control structures. For example, in the expression

```
f(if x > y then 2*x else 2*y)
```

a new expression instance is created for the evaluation of x > y. However, since the outcome of the if-then-else is the outcome of the selected clause (either 2*x or 2*y), the selected clause is treated as a subexpression in the surrounding expression instance. There are two expression instances for the above expression. One instance is used to isolate the evaluation of x > y from the evaluation occurring in the surrounding instance. Thus the above expression evaluates as either f(2*x) or f(2*y).

## 2.3 A Unified Syntax for Expression Evaluation

The syntax of Icon is designed to assist the user in the development of clear, understandable programs. This is accomplished by providing both mnemonic forms for various control mechanisms and a concise representation for expressions. Conciseness is obtained by providing an implicit representation for the goal-directed evaluation control mechanism. Goal-directed evaluation is such an inherent aspect of Icon expression evaluation that a visible syntax for it would be overbearing. The use of mnemonic forms for control mechanisms is intended to enhance program readability. Because this syntax is designed for program development, it is called 𝒫-syntax. Expressions written in 𝒫-syntax are referred to as 𝒫-expressions.

When discussing the expression evaluation mechanism in Icon, however, 𝒫-syntax obscures some of the underlying concepts. This section presents an alternative syntax for Icon. This syntax is intended as a descriptive device to simplify subsequent discussions of the semantics of the Icon expression evaluation mechanism. For this reason it is called ℰ-syntax. Expressions written in ℰ-syntax are referred to as ℰ-expressions.

### 2.3.1 Language Primitives

Literals, identifiers, and operators constitute the language *primitives* in Icon. Aside from syntactic considerations, there is nothing that distinguishes operators from function- and procedure-valued identifiers. The value of an operator is the function that the operator performs. In ℰ-syntax, operators, functions, and procedures all have the same syntax, and are collectively termed *functions*. Operators are represented by a name for the function that performs the indicated action. A function bound with its arguments is a *function call*. A function call is represented in ℰ-syntax as an $n$-tuple as in LISP, where the first element of the $n$-tuple is the function, and the remaining elements are the arguments of that function. For example, the 𝒫-expressions

```
write("hello world")
show(x,y,z)
1 + 2
1 to 5
```

are written respectively as the ℰ-expressions

(write, "hello world")  
(show, x, y, z)  
(*add*, 1, 2)  
(*to*, 1, 5)

The sans-serif italic font is used to distinguish functions from the names of Icon identifiers. For example, the initial value of the variable write is the function *write*.

### 2.3.2 Keywords

Keywords are expressions. The only keywords of interest here are the 𝔏-expressions &null and &fail. &null always evaluates to the *null value* and has the 𝔖-syntax *&null*. The null value plays a special role as the initial value of variables. Evaluation of &fail always fails to produce a result and has the 𝔖-syntax *&fail*.

### 2.3.3 Control Regimes

The 𝔏-syntax for control structures represent *control regimes*. A control regime specifies a particular method of expression evaluation. For example, sequential processing of expressions that are separated by semicolons is a control regime in which expressions are evaluated from left to right.

The 𝔖-syntax for control regimes is

*regime* : *arguments*

where *regime* is the name of a control regime and *arguments* is a list of expressions constituting the operands to that control regime. The names for specific control regimes are chosen to be indicative of the corresponding control structures, and are shown here in an Old English typeface. Thus the 𝔏-expression

if $e_1$ then $e_2$ else $e_3$

is represented in 𝔖-syntax as

𝔍𝔣 : $e_1,e_2,e_3$

and the 𝔏-expression

every $e_1$ do $e_2$

has 𝔖-syntax

𝔈𝔳𝔢𝔯𝔶 : $e_1,e_2$

Goal-directed evaluation, which has no explicit representation in 𝔏-syntax, is the control regime 𝔊𝔬𝔞𝔩 in 𝔖-syntax. 𝔊𝔬𝔞𝔩 is used in the evaluation of all functions. Thus the 𝔏-expression

1 to 10

has 𝔖-syntax

𝔊𝔬𝔞𝔩 : *to*,1,10

Arguments that are omitted in the 𝔏-syntax are denoted by $\phi$ in the 𝔖-syntax. Hence the 𝔖-expression

$\mathfrak{If}: e_1, e_2, \phi$

corresponds to the $\mathfrak{L}$-expression

if $e_1$ then $e_2$

Brackets provide any necessary grouping in the $\mathfrak{E}$-syntax, so that the $\mathfrak{L}$-expression

every 1 to 10

is written in $\mathfrak{E}$-syntax as

$\mathfrak{Every}: [\mathfrak{Goal}{:}\mathit{to}, 1, 10], \phi$

and the $\mathfrak{L}$-expression

if x > y then 1 to x

is the $\mathfrak{E}$-expression

$\mathfrak{If}: [\mathfrak{Goal}{:}\mathit{greaterthan}, x, y], [\mathfrak{Goal}{:}\mathit{to}, 1, x], \phi$

## 2.4 Result Sequences

$\mathfrak{E}$-syntax makes it easier to compare common components of a language by describing these components using a uniform syntax. $\mathfrak{E}$-syntax does not, in itself, provide much insight into the semantics of expression evaluation. Insight into the static aspects of expression evaluation can be gained from examining the sequences of results produced during expression evaluation. This section presents a notation for describing these *result sequences*.

A result sequence is defined as the sequence of results that an expression is capable of producing. The result sequence produced by the expression $e$ is denoted by

$$\mathfrak{S}(e) = \{s_1, s_2, \ldots, s_n\}$$

where $s_i$ is the $i$ th result generated during the evaluation of $e$. For example, the result sequence for 1 to 10 is

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

Subscripts are used to distinguish different expressions and their results. For example, the result sequences for expressions $e_1$ and $e_2$ are denoted by

$$\mathfrak{S}(e_1) = \{s_{1_1}, s_{1_2}, \ldots, s_{1_n}\}$$

$$\mathfrak{S}(e_2) = \{s_{2_1}, s_{2_2}, \ldots, s_{2_m}\}$$

The *empty sequence*, { }, is a sequence containing no elements and is denoted by $\Phi$. Another common sequence contains exactly one element, the null value. This sequence is denoted by $\Lambda$.

Subsequences are denoted using subscripts and superscripts, as in

$$\mathfrak{S}^j_i(e) = \{s_i, \ldots, s_{\min(j,n)}\}$$

If $i = 1$, the subsequence is abbreviated as

$$\mathfrak{S}_1^i(e) = \mathfrak{S}^i(e)$$

If $i = j$, the subsequence is abbreviated as

$$\mathfrak{S}_i^i(e) = \mathfrak{S}_i(e)$$

The *size* of a sequence is the number of elements it contains.

$$|\,\mathfrak{S}(e)\,| = n$$

where $0 \leq n \leq \infty$.

*Concatenation* of sequences is denoted by

$$\mathfrak{S}(e_1) \oplus \mathfrak{S}(e_2) = \{s_{1_1}, \ldots, s_{1_n}, s_{2_1}, \ldots, s_{2_m}\}$$

Concatenations of a sequence with itself are represented using exponential notation. For example,

$$\mathfrak{S}(e) \oplus \mathfrak{S}(e) = \mathfrak{S}(e)^2$$

$$\mathfrak{S}(e) \oplus \mathfrak{S}(e) \oplus \cdots = \mathfrak{S}(e)^\infty$$

The empty sequence is the identity element with respect to the concatenation of sequences:

$$\Phi \oplus \mathfrak{S}(e) = \mathfrak{S}(e) \oplus \Phi = \mathfrak{S}(e)$$

Concatenation of result sequences from a number of expressions, as in

$$\mathfrak{S}(e_1) \oplus \mathfrak{S}(e_2) \oplus \cdots \oplus \mathfrak{S}(e_n)$$

is denoted by

$$\mathop{\Xi}\limits_{i=1}^{n} \mathfrak{S}(e_i)$$

The *dot product* of two sequences produces a sequence of pairs as defined by

$$\mathfrak{S}(e_1) \cdot \mathfrak{S}(e_2) = \{(s_{1_1}, s_{2_1}), (s_{1_2}, s_{2_2}), \ldots, (s_{1_k}, s_{2_k})\}$$

where $k = \min(n, m)$. This can be generalized to the dot product of $n$ sequences to produce a sequence of $n$-tuples.

The *cross product* of two or more sequences also produces a sequence of $n$-tuples. For example, the cross product of two sequences produces a sequence of pairs as given by

$$\begin{aligned}\mathcal{S}(e_1)\times\mathcal{S}(e_2) = \{ &(s_{1_1},s_{2_1}),(s_{1_1},s_{2_2}),\ldots,(s_{1_1},s_{2_m}),\\ &(s_{1_2},s_{2_1}),(s_{1_2},s_{2_2}),\ldots,(s_{1_2},s_{2_m}),\\ &\ldots,\\ &(s_{1_n},s_{2_1}),(s_{1_n},s_{2_2}),\ldots,(s_{1_n},s_{2_m})\}\end{aligned}$$

Certain operations produce result sequences from sequence elements. If $\rho$ is such an operation, the *application* of an arbitrary operation $\rho$ to a sequence yields the concatenation of the sequences resulting from applying $\rho$ to every element in the original sequence. Application of $\rho$ to a sequence is denoted by $\rho::\mathcal{S}(e)$, and application of $\rho$ to a sequence element is denoted by $\rho{:}s$. That is

$$\begin{aligned}\rho::\mathcal{S}(e) &= \mathcal{S}(\rho{:}s_1) \oplus \mathcal{S}(\rho{:}s_2) \oplus \cdots \oplus \mathcal{S}(\rho{:}s_n)\\ &= \mathop{\Xi}_{i=1}^{n}\mathcal{S}(\rho{:}s_i)\end{aligned}$$

### 2.4.1 Result Sequences and Side Effects

Some expressions are not evaluated for their results, but rather for the side effects of their evaluation. For example, the function *write* produces its last argument as its result, but it is almost always used to produce output to a file. Furthermore, some operations rely upon side effects in order to work properly. The function *read*, for example, advances a file pointer each time it produces a result, so that subsequent evaluation produces a new result.

While side effects play a very real and important role in expression evaluation, they also tend to obscure more fundamental aspects of an expression evaluation mechanism. Result sequences are not intended to describe side effects, but rather to describe the static aspects of expression evaluation.

### 2.4.2 Result Sequences for Literals, Identifiers and Keywords

The result sequence for an identifier or literal consists of the identifier or the value of the literal. Hence the result sequences for x and 3 are {x} and {3}, respectively. The $\mathcal{E}$-expressions *&null* and *&fail* have the result sequences $\Lambda$ and $\Phi$, respectively.

### 2.4.3 Result Sequences for Functions

Functions are evaluated by the *function invocation* operation, $\Gamma$. $\Gamma$ is applied to a function call to produce a result sequence, and is represented in $\mathfrak{S}$-syntax as

$\Gamma$: *functioncall*

For example, the conjunction operator, &, has as its value the function *conj* that returns its right operand. Conjunction can be invoked as in the $\mathfrak{S}$-expression

$\Gamma$: (*conj*,x,y)

to produce the result sequence {y}.

In Icon, the application of $\Gamma$ is integrated into the operation of goal-directed evaluation. No attempt is made here to describe the actual evaluation of functions, except to note that $\Gamma$ produces a (possibly empty) sequence of results from a function call. For example, the result sequence for

$\Gamma$: (*to*,1,5)

is {1,2,3,4,5} and the result sequence for

$\Gamma$: (*greaterthan*,3,4)

is $\Phi$.

### 2.4.4 Result Sequences for Control Regimes

A control regime takes a list of arguments consisting of expressions and determines the order in which these arguments are evaluated. For example, one control regime may evaluate its arguments from left to right, while another may use the outcome of evaluating its first argument to select another argument to evaluate.

Like primitive operations, control regimes have result sequences. However, unlike primitive operations, control regimes deal with result sequences, not results. For example, the result sequence for

$\mathfrak{If}: e_0, e_1, e_2$

depends upon the size of the result sequence for $e_0$ and determines whether the result sequence for this expression is the result sequence for $e_1$ or the one for $e_2$.

The result sequence of a control regime can be used to help characterize the evaluation of that control regime. Describing the result sequence and its derivation for a control regime provides some insight into the semantics of that control regime. The result sequences for some representative control regimes in Icon are given below.

*Sequential Processing.* Sequential processing, represented by the $\mathfrak{L}$-expression

$$\{e_1;e_2;\cdots;e_n\}$$

and the $\mathfrak{S}$-expression

$$\mathfrak{Sequence} : e_1,e_2,\ldots,e_n$$

is a control regime that evaluates its arguments in order from left to right, limiting all the arguments except the rightmost one to at most one result. The result sequence for $\mathfrak{Sequence}$ is the result sequence of its rightmost argument. Hence

$$\mathfrak{S}(\mathfrak{Sequence}:e_1,e_2,\ldots,e_n) = \mathfrak{S}(e_n)$$

Note that the result sequence for $\mathfrak{Sequence}$ does not depend on the evaluation of $e_1,e_2,\ldots,e_{n-1}$.

*Goal-directed evaluation.* The goal-directed evaluation control regime forms the cross product of the result sequences for its arguments, constructing a sequence composed of function calls. Function invocation is then applied to the function calls to produce a new result sequence. Formally,

$$\mathfrak{S}(\mathfrak{Goal}:e_0,e_1,\ldots,e_n) = \Gamma::\mathfrak{S}(e_0)\times\mathfrak{S}(e_1)\times\cdots\times\mathfrak{S}(e_n)$$

For example,

$$\mathfrak{Goal} : \mathit{to},1,5$$

has the result sequence

$$\begin{aligned}\mathfrak{S}(\mathfrak{Goal}:\mathit{to},1,5) &= \Gamma::\{\mathit{to}\}\times\{1\}\times\{5\}\\ &= \Gamma::\{(\mathit{to},1,5)\}\\ &= \{\Gamma:(\mathit{to},1,5)\}\\ &= \{1,2,3,4,5\}\end{aligned}$$

Similarly, the $\mathfrak{L}$-expression

```
3 < (1 to 5)
```

is written in $\mathfrak{S}$-syntax as

$$\mathfrak{Goal} : \mathit{less},3,[\mathfrak{Goal}:\mathit{to},1,5]$$

From above,

$$\begin{aligned}\mathfrak{S}(\mathfrak{Goal}:\mathit{less},3,[\mathfrak{Goal}:\mathit{to},1,5]) &= \mathfrak{S}(\mathfrak{Goal}:\mathit{less},3,\{1,2,\ldots,5\})\\ &= \Gamma::\{\mathit{less}\}\times\{3\}\times\{1,2,\ldots,5\}\\ &= \Gamma::\{(\mathit{less},3,1),(\mathit{less},3,2),\ldots,(\mathit{less},3,5)\}\end{aligned}$$

$$= \{\Gamma{:}(less,3,1)\} \oplus \{\Gamma{:}(less,3,2)\} \oplus \cdots \oplus \{\Gamma{:}(less,3,5)\}$$

$$= \Phi \oplus \Phi \oplus \Phi \oplus \{4\} \oplus \{5\}$$

$$= \{4,5\}$$

As a final example of goal-directed evaluation, consider

```
(1 to 5) & x
```

with $\mathfrak{G}$-syntax

$$\mathfrak{Goal} : conj,[\mathfrak{Goal}{:}to,1,5],x$$

From above,

$$\mathfrak{S}(\mathfrak{Goal}{:}conj,[\mathfrak{Goal}{:}to,1,5],x) = \Gamma{::}\{conj\}\times\{1,2,3,4,5\}\times\{x\}$$

$$= \Gamma{::}\{(conj,1,x),(conj,2,x),\ldots,(conj,5,x)\}$$

$$= \{\Gamma{:}(conj,1,x)\} \oplus \{\Gamma{:}(conj,2,x)\} \oplus \cdots \oplus \{\Gamma{:}(conj,5,x)\}$$

$$= \{x,x,x,x,x\}$$

*Alternation.* Alternation, $e_1 \mid e_2$, is the control regime $\mathfrak{Alternation}$ that simply concatenates the result sequences for its arguments. That is

$$\mathfrak{S}(\mathfrak{Alternation}{:}e_1,e_2) = \mathfrak{S}(e_1) \oplus \mathfrak{S}(e_2)$$

*If-then-else.* The $\mathfrak{If}$ control regime uses the size of the result sequence of its first argument to determine which of the two remaining arguments, or *arms*, to evaluate. The result sequence for $\mathfrak{If}$ is the result sequence for the selected arm. That is

$$\mathfrak{S}(\mathfrak{If}{:}e_0,e_1,e_2) = \begin{array}{ll} \mathfrak{S}(e_1) & \text{if } \mathfrak{S}(e_0) \neq \Phi \\ \mathfrak{S}(e_2) & \text{otherwise} \end{array}$$

*Outcome inversion.* $\mathfrak{Invert}$ is the outcome inversion control regime denoted by the $\mathfrak{L}$-expression

not $e$

and the $\mathfrak{G}$-expression

$$\mathfrak{Invert} : e$$

The result sequence for $\mathfrak{Invert}$ is

$$\mathfrak{S}(\mathfrak{Invert}{:}e) = \begin{array}{ll} \Lambda & \text{if } \mathfrak{S}(e) = \Phi \\ \Phi & \text{otherwise} \end{array}$$

That is, the result sequence for $\mathfrak{Invert}$ is empty if its argument is the empty sequence, and $\Lambda$ otherwise.

*Repeated evaluation.* The repeated evaluation control regime, with $\mathfrak{L}$-syntax

$$|e$$

and $\mathfrak{S}$-syntax

$$\mathfrak{Reval}: e$$

has the result sequence

$$\mathfrak{S}(\mathfrak{Reval}{:}e) = \mathfrak{S}(e)^{\infty}$$

In practice, side effects are relied upon to limit the size of the result sequence for $\mathfrak{Reval}$. If $\mathfrak{S}(e)_i$ represents the $i$th term in the expansion of $\mathfrak{S}(e)^{\infty}$ and $\mathfrak{S}(e)_i$ has the result sequence $\Phi$, then evaluation of $\mathfrak{Reval}$ terminates after evaluating this term. Unless $\mathfrak{S}(e)_1$ is $\Phi$, only side effects can cause $\mathfrak{S}(e)_i$ to be $\Phi$ for $i > 1$.

*Limitation.* Limitation, with $\mathfrak{L}$-syntax

$$e_1 \setminus e_2$$

and $\mathfrak{S}$-syntax

$$\mathfrak{Limit}{:}e_1, e_2$$

has the result sequence

$$\mathfrak{S}(\mathfrak{Limit}{:}e_1, e_2) = \mathop{\Xi}_{i=1}^{|\mathfrak{S}(e_2)|} \mathfrak{S}^{s_{2_i}}(e_1)$$

Note that if $|\mathfrak{S}(e_2)| = 1$ and $s_{2_1} = k$, then the result sequence for limitation is simply a subsequence of the result sequence for $e_1$. That is,

$$\mathfrak{S}(\mathfrak{Limit}{:}e_1, e_2) = \mathfrak{S}^k(e_1)$$

*Iteration.* The iteration control regimes $\mathfrak{Every}$, $\mathfrak{While}$, $\mathfrak{Repeat}$, and $\mathfrak{Until}$ are evaluated for side effects. Hence the iteration control regimes produce the empty result sequence $\Phi$.

$$\mathfrak{S}(\mathfrak{Every}{:}e_1, e_2) = \mathfrak{S}(\mathfrak{While}{:}e_1, e_2) = \mathfrak{S}(\mathfrak{Repeat}{:}e_1) = \mathfrak{S}(\mathfrak{Until}{:}e_1, e_2) = \Phi$$

The iteration control regimes are all *equivalent*, in that $\mathfrak{Every}$ can be used to formulate the other iteration control regimes. Furthermore, $\mathfrak{Every}$ can be constructed from $\mathfrak{Goal}$ and a few other control regimes. These and other semantic equivalences are described below.

## 2.5 Supplementary Control Regimes

The following control regimes are constructed from the preceding control regimes and have no direct $\mathfrak{I}$-syntax. They are described here to provide a convenient notation for the description of subsequent control regimes.

### 2.5.1 Limitation to Exactly One Result

$\mathfrak{One}$ is a control regime that produces exactly one result. $\mathfrak{One}$ : $e$ is equivalent to

```
( e | &null ) \ 1
```

or, in $\mathfrak{G}$-syntax

$\mathfrak{Limit}$ : [$\mathfrak{Alternation}$:$e$ ,*&null* ], 1

Alternation is needed to guarantee that the result sequence for $\mathfrak{One}$ has at least one element, since $\mathfrak{S}(e)$ may be $\Phi$. Limitation provides the first result from this sequence, so that

$$| \mathfrak{S}(\mathfrak{One}:e) | = 1$$

### 2.5.2 Repeated First Result

$\mathfrak{First}$ is similar to $\mathfrak{Reval}$ except that the argument to $\mathfrak{First}$ is limited to at most one result. As with $\mathfrak{Reval}$, evaluation of $\mathfrak{First}$ terminates if its argument fails to produce a result. $\mathfrak{First}$ : $e$ is equivalent to the $\mathfrak{I}$-expression

```
| (e \ 1)
```

and the $\mathfrak{G}$-expression

$\mathfrak{Reval}$ : [$\mathfrak{Limit}$:$e$ , 1]

$\mathfrak{First}$ has a result sequence consisting of an infinite repetition of the first element in the result sequence for its argument. That is, the result sequence for $\mathfrak{First}$ is given by

$$\mathfrak{S}(\mathfrak{First}:e) = \mathfrak{S}_1(e)^{\infty}$$

subject to the same termination conventions as $\mathfrak{Reval}$.

### 2.5.3 Multiple Conjunction

The conjunction operator can be used to provide mutual goal-directed evaluation of two expressions. This generalizes to any number of expressions, but quickly becomes notationally unwieldy in $\mathfrak{S}$-syntax. For example, the $\mathfrak{L}$-expression

$$e_1 \& e_2 \& e_3 \& e_4$$

has the equivalent $\mathfrak{S}$-expression

$$\mathfrak{Goal} : \mathit{conj}, e_1, [\mathfrak{Goal}:\mathit{conj}, e_2, [\mathfrak{Goal}:\mathit{conj}, e_3, e_4]\,]$$

The function *mulconj* is an $n$-ary function that can be used to replace the multiple use of *conj* in situations where the $\mathfrak{S}$-syntax becomes unwieldy. The function *mulconj* produces its rightmost argument. That is

$$\mathfrak{S}(\Gamma : \mathit{mulconj}, x, y, z) = \{z\}$$

For example, the $\mathfrak{L}$-expression given above can also be represented by the $\mathfrak{S}$-expression

$$\mathfrak{Goal} : \mathit{mulconj}, e_1, e_2, e_3, e_4$$

The function *mulconj* extends naturally into the *mutual goal-directed evaluation* function *mgde* with $\mathfrak{L}$-syntax

$$e_0(e_1, \ldots, e_n)$$

where $e_0$ evaluates to an integer instead of a procedure. The $\mathfrak{S}$-syntax more accurately reflects the fact that *mgde* is simply another function. The $\mathfrak{S}$-syntax is

$$\Gamma : \mathit{mgde}, e_0, \ldots, e_n$$

The integer value of $e_0$ determines which of the remaining arguments produces the result of *mgde*. For example,

$$\mathfrak{S}(\Gamma : (\mathit{mgde}, 3, x, y, z)) = \{z\}$$

## 2.6 Equivalences among Control Regimes

Some control regimes can be expressed in terms of others. For example, the supplementary control regimes given above were formulated using the control regimes **Goal**, **If**, **Alternation**, **Reval**, and **Limit**.

These five control regimes are sufficient, in combination with *conj*, *&fail* and *&null*, to describe other control regimes. The control regimes listed above are referred to as the *basic* control regimes.

### 2.6.1 Sequential Processing

**Sequence** evaluates its arguments from left to right and has the result sequence of the last argument. Because all the arguments except the last have no effect on the outcome of **Sequence**, they are be limited to exactly one result. Thus

**Sequence** : $e_1, \ldots, e_{n-1}, e_n$

may be expressed as

**Goal** : *mulconj*,[**One**:$e_1$], . . . , [**One**:$e_{n-1}$], $e_n$

### 2.6.2 Outcome Inversion

**Invert**:$e$ may be expressed as

**If** : $e$ ,*&fail*,*&null*

### 2.6.3 Iteration Regimes

All of the iteration regimes can be expressed in terms of the basic regimes. However, the approach is simplified by first expressing the iteration regimes in terms of each other.

**Repeat** only terminates because of side effects. That is,

**Repeat** : $e$

is expressible as

**While** : [**One**:$e$ ],$\phi$

where **One** prevents failure in the evaluation of $e$ from terminating the evaluation of **While**. Similarly, Until : $e_1$,$e_2$

is expressed as

While : [Invert:$e_1$],$e_2$

Expressing While in terms of Every is simplified by the use of First. That is,

While : $e_1$,$e_2$

is expressed as

Every : [First:$e_1$],$e_2$

or

Goal : *mulconj*,[First:$e_1$],[Limit:$e_2$,1],*& fail*

Finally,

Every : $e_1$,$e_2$

can be expressed

Goal : *mulconj*,$e_1$,[Limit:$e_2$,1],*& fail*

It is interesting to note the similarity between the basic regime formulation for While and the formulation for Every. The only difference between the two is that the first argument to While repeatedly produces its first result, while the first argument to Every produces its entire result sequence.

## 2.7 Additional Control Regimes

Generating expressions are notationally concise and have a great deal of computational power. However, the lack of sufficient 'generator-based' control regimes often limits the use of generators. For example, the expression

```
every f(!alist)
```

invokes f on every element of alist, but to invoke f on just the even-numbered elements, or on the tenth through twentieth elements, requires a radically different expression such as

```
every f(alist[2 to *alist by 2]
```

Result sequences suggest a number of interesting control regimes that provide additional control over generators. This section describes several of these control regimes and presents possible ℒ-syntax forms for these regimes.

The control regime Subsequence with 𝒮-syntax

$\mathfrak{Subsequence} : e_1, e_2, e_3$

and proposed $\mathfrak{L}$-syntax

$e_1 \backslash [e_2 : e_3]$

has a result sequence that is a subsequence of the result sequence for its first argument. For example,

$\mathfrak{S}(\mathfrak{Subsequence}{:}e_1, 5, 10) = \mathfrak{S}_5^{10}(e_1)$

Note that $\mathfrak{Subsequence}$ is a generalization of $\mathfrak{Limit}$. That is,

$\mathfrak{Limit} : e_1, e_2 = \mathfrak{Subsequence} : e_1, 1, e_2$

The $\mathfrak{L}$-syntax

$e_1 \backslash [e_2{:}0]$

represents the proposed control regime given by the $\mathfrak{E}$-expression

$\mathfrak{Subsequence} : e_1, e_2, \infty$

so that

```
(1 to 10) \ [7:0]
```

has the result sequence {7, 8, 9, 10}. Finally,

```
every f(!alist \ [10:20])
```

invokes f on the tenth through twentieth elements of alist.

A generalization of $\mathfrak{Subsequence}$ is $\mathfrak{Newsequence}$ with $\mathfrak{E}$-syntax

$\mathfrak{Newsequence} : e_1, e_2$

and proposed $\mathfrak{L}$-syntax

$e_1 \backslash\backslash\ e_2$

If the result sequence for $e_2$ consists of integers in strictly increasing order, the result sequence for $\mathfrak{Newsequence}$ is composed of those elements from the result sequence for $e_1$ that are indexed by the values of the elements of the result sequence for $e_2$. Formally,

$$\mathfrak{S}(\mathfrak{Newsequence}{:}e_1, e_2) = \mathfrak{S}_{s_{2_1}}(e_1) \oplus \mathfrak{S}_{s_{2_2}}(e_1) \oplus \cdots \oplus \mathfrak{S}_{s_{2_m}}(e_1)$$

For example, the $\mathfrak{L}$-expression

$e_1 \backslash\backslash\ (e_2 \text{ to } e_3)$

is the same as

$$e_1 \setminus [e_2:e_3]$$

except when the result of $e_3$ is 0.

As a final example,

```
every f(!alist  \ \ (2 to *alist by 2))
```

invokes f on the even-indexed elements of alist.

# Chapter 3

# Co-Expressions

Generators in Icon are limited by the syntax of the language. This has the advantage of providing straightforward means of controlling generators, as well as permitting efficient implementation. Further, the 'first-in, last-out' activation of nested generators makes generators well suited to combinatorial applications (Griswold, Hanson, and Korb 1981).

The evaluation of a generator is restricted to a single lexical site within a program, however. Furthermore, every evaluation of a generator produces elements from the result sequence for that generator *from the first element on.* For example, if alist is a list, then !alist is a generator that produces the elements in the list. However, there is no straightforward way to use this generator to print, for example, only every other element in alist, because there is no way to evaluate !alist at several sites within a program, without reproducing the result sequence from the beginning.

This chapter describes a mechanism that frees the evaluation of a generator from its lexical site. Freeing the evaluation of a generator allows the programmer to access the elements of the result sequence for that generator as needed, where needed. This makes it possible to write clearer, more concise programs in many situations. In addition, this mechanism provides facilities at the expression level for developing evaluation strategies similar to those provided at the procedure level in languages with coroutines.

## 3.1 Co-Expression Creation and Activation

The expression

create *expr*

creates a *co-expression* containing *expr*. A co-expression is a data object consisting of an expression and an environment in which to evaluate that expression. This environment is a copy of the environment in which the create is performed and includes copies of any dynamic local identifiers (as opposed to static local identifiers) referenced by the expression. As such, the co-expression contains the state information necessary for the evaluation of the expression, independent of surrounding context.

Unlike conventional expressions, which are evaluated in environments that are lexically restricted to fixed locations in a program, an expression within a co-expression may be *activated* wherever a result is desired from the sequence produced by the evaluation of the expression. The expression

```
@x
```

activates x to obtain the next result from the result sequence for the co-expression.

For example, evaluation of the following expression assigns to the identifier x a co-expression for producing the elements of alist.

```
x := create !alist
```

Activations of x produce successive elements from alist. For example

```
while write(@x)
```

writes all the elements of alist, and writing the even-numbered elements of alist may be accomplished with

```
while @x do
   write(@x)
```

Activation of a co-expression fails once its result sequence has been generated. Subsequent attempts to activate the same co-expression also fail. Hence, in the above examples, activation of x fails after all the elements of alist have been generated.

The activation operator itself is limited to at most one result. Hence, only one result is produced by the expression

```
every write(@x)
```

However, repeated evaluation may be applied to a co-expression to achieve the effect of 'unlimiting' activation. Repeated evaluation is effectively a means of obtaining results from a looping expression. If that expression is a co-expression activation, then repeated evaluation iterates over the result sequence of the co-expression. For example, the following expression writes all the elements of alist.

```
every write(|@x)
```

As another example, consider the generator find(s1,s2). Creating a co-expression for find(s1,s2) permits the elements of the result sequence produced by evaluation of find to be obtained when and where they are needed. Hence

```
x := create find("ab", "abracadabra")
write("The first is at ", @x)
write("The second is at ", @x)
```

outputs

```
The first is at 1
The second is at 8
```

Note that without co-expressions find(s1,s2) is reevaluated each time it occurs. Thus

```
write("The first is at ",find("ab", "abracadabra"))
write("The second is at ",find("ab", "abracadabra"))
```

outputs

```
The first is at 1
The second is at 1
```

## 3.2 Operations on Co-Expressions

If x is a co-expression, then

```
*x
```

is the number of results that have been produced from the result sequence for x. For example,

```
x := create find("text",!file)
while write(@x)
count := *x
```

outputs the column positions of the string text in file, and assigns to count the number of occurrences of text.

The *refresh* operation, ^x, returns a copy of the co-expression x with the environment portion being the same as when x was created. Thus the refresh operation provides a means of repeating the result sequence of a co-expression. For example,

```
x := create find("ab", "abracadabra")
write("The first is at ", @x)
write("The second is at ", @x)
x := ^x
write("The first is still at ", @x)
```

outputs

```
The first is at 1
The second is at 8
The first is still at 1
```

## 3.3 Generators and Co-Expressions

One of the simplest uses of co-expressions is in forming *unbounded* selection operations whose scopes are not limited to a single site of evaluation. For example, the following code segment decollates a list, alist, into two lists, odd and even. The elements of odd are the elements of alist with odd indices, and the elements of even are those with even indices.

```
blist := create !alist
odd := list()
even := list()

every |(put(odd, @blist) | put(even, @blist))
```

A more succinct form of the above every expression is

```
every |put(odd | even, @blist)
```

Similarly, string decollation can be accomplished with

```
s1 := s2 := ""
nextchar := create !s
every |((s1 | s2) ||:= @nextchar)
```

Because co-expressions exist indefinitely, many algorithms can use generators that could not without co-expressions.

Consider a procedure for supplying successive integer values. It is simple and natural to express this as a generator, as in

```
procedure incr(n)
   repeat {
      suspend n
      n +:= 1
      }
end
```

Creating separate co-expressions for several different invocations of incr permits use of this generator within several independent co-expressions. For example, incr can be used to create a label generator that generates successive labels of the form L*nnn* starting with L010.

```
genlab := create ("L" || right(incr(10), 3, "0"))
```

At the same time, a second co-expression can use incr as in

```
nextint := create (incr(0) % maxcycle)
```

to repeatedly cycle through a sequence of integers.

Linked lists provide an example in which the generation of elements is more complicated. Because there is no linked list datatype in Icon, they must be simulated using existing datatypes. The nodes of a linked list can be represented with records declared by

```
record lnode(value, link)
```

A generator for sequencing through elements of a linked list is shown below. Creating a co-expression that invokes this procedure results in an unbounded selection operation for generating the elements from a linked list.

```
procedure nextelement(llist)
   repeat {
      suspend llist.value
      llist := \llist.link | fail
      }
end
```

(In Icon, the expression \x succeeds only if the value of x is non-null.)

Co-expressions permit the separation of an algorithm from the situations in which it is to be used. This generally results in clearer, more concise code. For example, there are many applications, such as the 'same fringe' problem (Hewitt and Patterson 1970) that require access to the leaves of a binary tree. If the nodes of a binary tree are represented with records declared by

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

then the procedure

```
procedure leaves(t)
   if /t.ltree & /t.rtree then return t.data
   suspend leaves(\t.ltree | \t.rtree)
end
```

generates the leaves of the tree. The operation of the procedure depends upon the fact that node fields have null values until another value is assigned. (The expression /x succeeds if the value of x is null.)

The procedure leaves may be used in any application that requires access to the values of the leaves of a tree in sequence, as in

```
every write(leaves(tree))
```

By creating a co-expression for an invocation of leaves, this same procedure may be used as an unbounded selection operation. For example, the following code is equivalent to the every expression given above.

```
nextleaf := create leaves(tree)
   .
   .
   .
while write(@nextleaf)
```

In turn, unbounded selection permits the procedure to be used in more complex situations, such as in the procedure compare given in the next section.

## 3.4 Co-Expressions and the Evaluation of Multiple Generators

Goal-directed evaluation provides a *cross-product* form of analysis when several generators are present in the same expression (Griswold, Hanson and Korb 1981). This cross-product analysis is effectively a depth-first search for results among a set of possible results. While goal-directed evaluation is extremely useful in combinatorial applications, it provides little assistance in situations where the results need to be interleaved. By permitting the order of evaluation of generators to be specified by the programmer, co-expressions provide the capability of arbitrarily interleaving the results of generators.

A procedure may activate two or more co-expressions in parallel, providing *dot-product* analysis to complement the cross-product analysis provided by simple goal-directed evaluation.

For example, the following procedure determines if two co-expressions produce equivalent result sequences, assuming that the sizes of the two result sequences are the same. (A more general solution is presented in Chapter5.)

```
procedure compare(cx1, cx2)
local r1, r2
   while r1 := @cx1 & r2 := @cx2 do
      if r1 ~=== r2 then fail
   return
end
```

This procedure can be used in a solution to the same-fringe problem to walk two trees in parallel to determine if their leaf nodes have the same values in the same order:

```
if compare(create leaves(tree1), create leaves(tree2)) then
   write("same fringe")
else
   write("different fringe")
```

In addition to this simple form of dot-product analysis, co-expressions can be used with repeated evaluation to interleave the result sequences from two or more expressions. The result sequence for

```
|(@e1 | @e2)
```

consists of alternating results from e1 and e2. If activation of either co-expression fails, the remaining co-expression continues to produce results until its activation also fails. An example of interleaving results is the procedure merge that interleaves the characters from two strings.

```
procedure merge(s1, s2)
local e1, e2, s
   e1 := create !s1
   e2 := create !s2
   s := ""
   every s ||:= |(@e1 | @e2)
   return s
end
```

If the strings are of equal length, then merge collates the two strings. If the strings differ in length, then the extra characters in the longer string are appended to the resulting string. This approach is straightfoward, although there are more efficient methods using character sets and string mapping techniques (Griswold 1980a).

The technique of using repeated evaluation to interleave activations of co-expressions generalizes to any number of co-expressions. For example, code for interleaving four strings is obtained by

```
|(@e1 | @e2 | @e3 | @e4)
```

### 3.5 Co-Expressions as Coroutines

There is a close correspondence between semi-coroutine systems (Dahl 1972) and the capabilities of co-expressions that have been described in the previous sections. In a semi-coroutine system, program control can be passed between some master process and a number of subordinate coroutine processes. The subordinate processes may not pass control among themselves, however.

Activation of a co-expression *interrupts* evaluation of the activating expression and *continues* evaluation of the co-expression. Suspension from a co-expression interrupts evaluation of the co-expression and continues evaluation of the activating expression. Thus co-expressions and generators represent primitives from which semi-coroutines can be constructed.

Besides this semi-coroutine method of evaluation, a co-expression can also activate other co-expressions, producing a general coroutine style of evaluation. The effect of one co-expression activating another is simply that evaluation is interrupted in the first, and continued in the second, thus providing capabilities at the expression level that are similar to the capabilities provided by coroutines at the procedure level in languages such as SL5 (Hanson and Griswold 1978), and ACL (Marlin 1980).

#### 3.5.1 Built-in Co-Expressions

There are two co-expressions provided by the Icon system as aids in the use of co-expressions in a general coroutine style. These co-expressions are represented by the keywords &main and &source.

Program execution in Icon is initiated by an implicit call to the procedure main. The keyword &main is a co-expression for this call. Activation of &main from any co-expression returns control to the point of interruption in the evaluation of the call to main.

&source is a co-expression for the activating expression of the currently active co-expression. Control can be explicitly transferred from a co-expression to its activating expression by activating &source.

Access to &main and &source permits *any* co-expression to transfer control to any other co-expression, providing a general coroutine facility.

#### 3.5.2 Providing Results to Co-Expressions

A result can be supplied to the activation of a co-expression by

*expr1* @ *expr2*

which supplies the result of *expr1* to the activation of the co-expression that is the result of *expr2*. (This result is ignored if the co-expression is being activated for the first time.)

### 3.5.3 Examples of Co-Expressions as Coroutines

The following problem was originally posed by Grune (1977) to illustrate a number of coroutine facilities.

"Let A be a process that copies characters from some input to some output, replacing all occurrences of aa with b, and a similar process, B, that converts bb into c. Connect these processes in series by feeding the output of A into B."

Using co-expressions, this problem can be solved as follows.

```
global A, B

procedure main()
   A := create compress("a", "b", create |reads(), B)
   B := create compress("b", "c", A, &main)
   while writes(@B)
end

procedure compress(c1, c2, in, out)
local ch
   repeat {
      ch := @in
      if ch == c1 then {
         ch := @in
         if ch == c1 then
            ch := c2
         else
            c1 @ out
         }
      ch @ out
      }
end
```

This solution is similar to a solution originally presented in Simula (Lynning 1978) and translated into ACL by Marlin (1980). Like their solutions and those proposed by Grune, it assumes an infinite stream of input. Like these solutions, the one above creates two instances of the same procedure for the operation of both A and B. The Icon version is simplified slightly by the ability to transfer results between co-expressions, however.

The following example uses co-expressions to implement the Sieve of Erastothanes. The technique is based upon a similar one used to illustrate a use of coroutines (McIlroy 1968) and filtered variables (Hanson 1978).

The sieve supplies an infinite stream of integers through a cascade of 'filters', each of which checks to see if the integer is divisible by a specific known prime. Each filter activates the next filter in the cascade if the integer passes its test. If a filter finds an integer that is a multiple of its prime, the filter activates the source of integers and the cascade is restarted on the next integer. If the integer passes through the entire set of filters successfully, it is output as a prime and a new filter is added to the cascade to test subsequent integers against this prime.

```
global num, cascade, source, nextfilter

procedure main()
   cascade := list()
   source := create              # root of sieve
               every num := 2 to huge_number do {
                  nextfilter := create !cascade          # sequence of filters
                  @@nextfilter          # get first filter and activate it
                  }
   push(cascade, create sink())         # sink starts as only filter
   @source                       # start the sieve
end

procedure sink()
local prime
   repeat {
      write(prime := num)
      push(cascade, create filter(prime))                # add filter to cascade
      @source                    # start processing next number
      }
end

procedure filter(prime)
   repeat
     if num % prime = 0 then @source          # try next num
     else @@nextfilter           # get next filter and activate it.
end
```

The co-expression source generates the integers and starts the cascade on each integer. Each filter in the cascade is a co-expression testing the potential prime against a specific known prime. The co-expression sink processes new primes and is always the last filter in the cascade. An additional co-expression is used to sequence through the filters in cascade using the selection operator. Note that each filter is invoked exactly once. From then on, control is simply passed between source and the various filters (including sink).

Actually, there is no need for any of the procedures other than main. This example can be written as

```
global num, cascade, source, nextfilter

procedure main()
local prime
   cascade := list()
   source := create {
                every num := 2 to huge_number do
                   @@(nextfilter := create !cascade)
                @&main
                }
   push(cascade, create
         repeat {
            write(prime := num)
            push(cascade, create repeat
                  if num % prime = 0 then @source
                  else @@nextfilter)
            @source
            })
   @source
end
```

This version does not show the logical division of the algorithm as well as the previous version, however, and works properly only because co-expressions maintain their own copies of local identifiers.

# Chapter 4

# Implementation

While generating expressions are a fundamental aspect of Icon, their use is not restricted to Icon. The language Cg (Budd 1981), which integrates generators and goal-directed evaluation into the C programming language, demonstrates that generators can extend more conventional languages. The integration of generators and goal-directed evaluation into languages with stack-based implementations can be accomplished in a straightforward and efficient manner.

Implementation of generators and co-expressions consists of two parts: run-time support and generated code. The run-time support includes the primitive actions needed for the evaluation of generators and co-expressions. The generated code organizes these primitives into control regimes.

This chapter presents models for the implementations of generators and co-expressions. The first model of generating expressions provides the most straightforward implementation, but requires the use of two physically distinct stacks. The second model merges the two stacks of the first model into a single stack, providing greater efficiency and simplifying the implementation of co-expressions. The model given for the implementation of co-expressions is based on the second model for generator evaluation. An interpreter based upon the second model has been written in Icon for a subset of Icon and appears in Appendices A and B. Appendix A contains a recursive descent parser for translating $\mathfrak{L}$-syntax into an intermediate code. Appendix B contains an interpreter for this intermediate code.

The second model of generating expressions and the corresponding model of co-expressions are compared to the actual implementation of these features in Icon and Cg. Appendix C contains an intermediate code interpreter more closely matching an actual implementation. Restrictions imposed on the actual implementations by scoping conventions and machine architecture are also discussed.

## 4.1 Implementation of Expression Instances

Expression instances in Algol-like languages are relatively uninteresting, amounting to little more than storage for temporary values that are used during evaluation of the expression. Furthermore, because expressions in these languages produce exactly one result, expression instances only exist while they are either passive or active. Thus a single stack can be used to maintain expression instances. In Icon, however, expression instances must contain more information concerning the state of the evaluation of that instance, and an Algol-like stack is no longer sufficient to maintain expression instances.

While the information associated with an expression instance may vary depending upon language features and implementation techniques, information typically associated with Icon expression instances includes:

1. A *passive instance pointer* pointing to a linked list of enclosing passive expression instances.

2. An *inactive instance pointer* pointing to a linked list of inactive subexpressions.

3. An *activation address* acting as a pointer into the program code. The activation address assumes different meanings depending upon the state of the expression instance and is explained in more detail later.

4. An *expression stack* used to hold any temporary results created during evaluation of the expression.

The first three items are referred to collectively as the *expression marker*. In practice, expression stacks are not separate entities, but simply represent areas on some *system stack* that are separated by expression markers. Nevertheless, it is convenient to view the system stack as a stack of expression instances, with each expression instance maintaining its own expression stack.

Expression instances are coalesced by factoring the expression markers out to an enclosing expression instance. Thus subexpressions correspond to expressions that have had their expression markers factored to some enclosing expression instance.

**4.1.1 Operations on Expression Instances**

Various implementation schemes for generators can be formulated in terms of operations upon expression instances. Icon program segments are used here to model several schemes.

Besides the standard features of Icon, the following operations involving expressions instances are assumed to be features of the implementation language.

1. Creating an expression instance is accomplished by

   ```
   create_instance()
   ```

   which returns a created expression instance of the form

| | |
|---|---|
| passive | |
| inactive | |
| save_pc | |
| estack | |

   The field labelled passive holds the passive instance pointer for that expression. The field labelled inactive holds the inactive instance pointer. The field labelled save_pc holds the activation address. Finally, estack is the expression stack area. Changes to this form dictated by different implementation approaches are indicated where appropriate.

2. An expression instance, i, is copied by

   ```
   copy(i)
   ```

3. The fields of an expression instance are accessed using the field reference operator of Icon. That is,

   ```
   i.passive := &null
   ```

   clears the passive instance pointer field of instance i, and

   ```
   \i.inactive
   ```

   succeeds if the inactive instance pointer of i is non-null.

4. Expression instances can be manipulated as data-objects. Because expression instances are maintained on stacks (most notably the system stack), pushing and popping them on to and off of a stack are accomplished with

   ```
   push(stack,i)
   ```

   and

   ```
   pop(stack)
   ```

   respectively. Another useful stack operation is

   ```
   popto(stack,object)
   ```

   which pops stack so that object is on top of the stack, and fails if object is not on the stack. The stack is left unchanged if popto fails.

5. Finally, the global identifiers pc and active are the machine location counter and a pointer to the currently active expression instance, respectively.

## 4.2 Goal-Directed Evaluation

Any temporary results that are present when a subexpression produces a result are restored when that subexpression is reactivated. For example, in the expression

```
5 + (1 to 10) > x
```

the value 5 is present as a temporary result in the currently active expression instance when the subexpression 1 to 10 produces a result. The addition operation replaces both 5 and the result produced by 1 to 10 with their sum. However, if 1 to 10 is reactivated to produce a subsequent result, 5 must be present in the active instance in order for evaluation to proceed properly. The information necessary to continue processing in a reactivated expression corresponds precisely to the information maintained as part of the expression instance for that expression.

When an expression is to be evaluated, an expression instance is created for that expression and evaluation proceeds within that instance. When a subexpression produces a result, a copy of the active expression instance is saved as an inactive expression instance and evaluation proceeds using the produced result. As an implementation optimization, copies of the active expression instance are saved only when the subexpression that has produced the result has the potential of producing subsequent results. This is accomplished by making each subexpression responsible for saving the active expression instance whenever that subexpression produces a result and is capable of producing additional results.

If failure occurs during evaluation, the currently active expression instance is destroyed and the most recently inactivated copy of that expression instance is activated. If there are no inactive copies of that expression instance, failure occurs in the evaluation of the enclosing instance.

When evaluation of an expression produces a result (as opposed to a subexpression producing a result), the result is provided to any enclosing instance. The currently active expression instance is destroyed, as are any inactive instances of that expression. The enclosing expression instance becomes the active expression instance.

### 4.2.1 The Two-Stack Model of Goal-Directed Evaluation

In the original implementation of Icon, two physically distinct stacks are used to implement goal-directed evaluation (Korb 1979). All expression instances that exist, but that are not inactive, are maintained on a *system* stack, denoted SYSSTK. The currently active expression instance is on the top of SYSSTK. The second stack, or *control* stack, is used to store inactive expression instances. The control stack is denoted CTLSTK.

There is no need for the passive instance pointer in expression instances, since passive expression instances are maintained in proper order on SYSSTK.

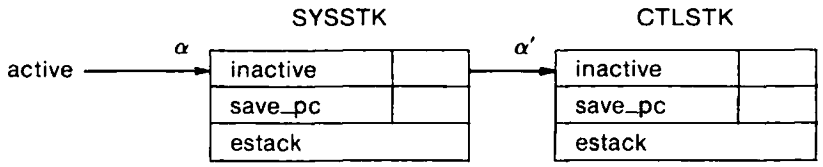
The activation address for expression instances on SYSSTK is the address to which program control is transferred whenever failure of a subexpression to that instance occurs. For instances on CTLSTK, the activation address is the address at which evaluation is to resume if the instance is reactivated.

There are three routines used to implement goal-directed evaluation:

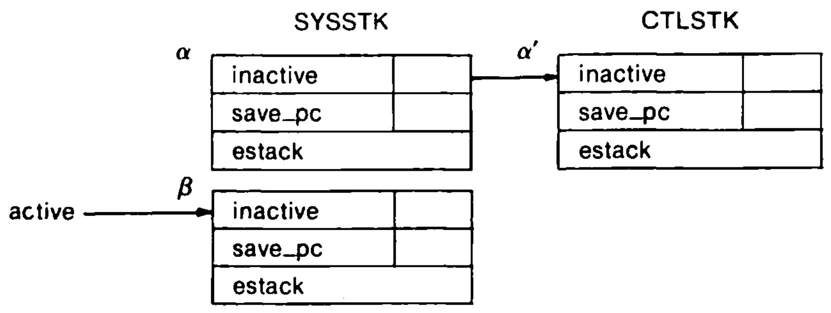
1. The procedure **mark** creates a new expression instance on SYSSTK and activates it.

```
procedure mark(failure_lab)
   push(SYSSTK, create_instance())
   top(SYSSTK).save_pc := failure_lab
   active := top(SYSSTK)
end
```

If SYSSTK and CTLSTK prior to the call of mark are



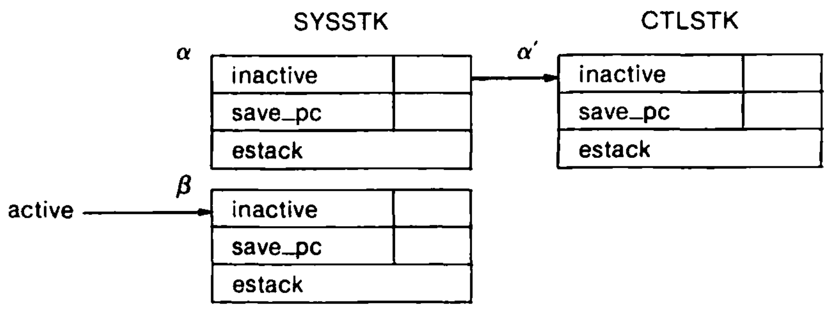
then after the call they are



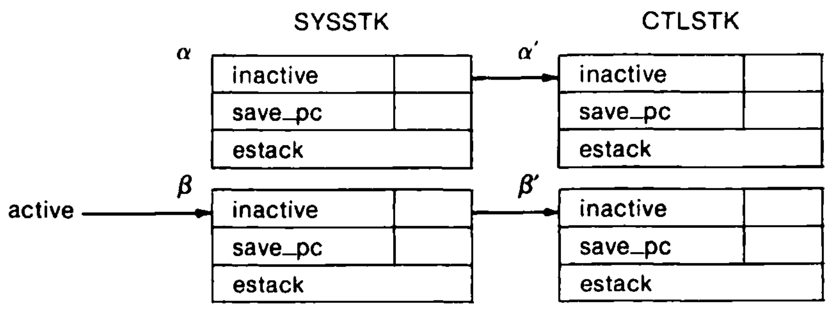
2. The procedure save saves a copy of the currently active expression instance on CTLSTK.

```
procedure save()
  push(CTLSTK, copy(active))
  top(CTLSTK).inactive := active.inactive
  top(CTLSTK).save_pc := pc
  active.inactive := top(CTLSTK)
end
```

If SYSSTK and CTLSTK before the call to save are



after the call they are



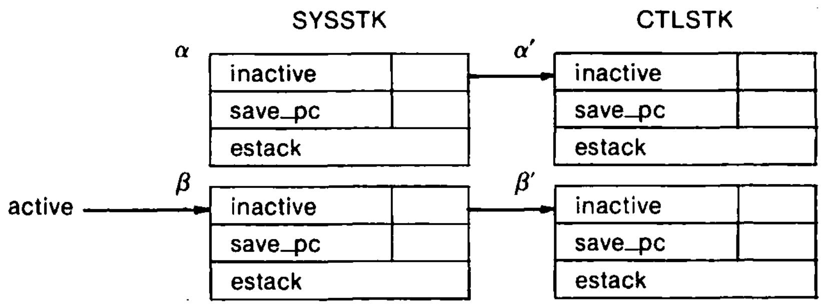
The result of the subexpression is then pushed onto the expression stack for the active instance and processing continues after the call of save().

3. The procedure drive handles success or failure of expression evaluation and depends upon the use of a global flag variable failure to signal success or failure of evaluation.

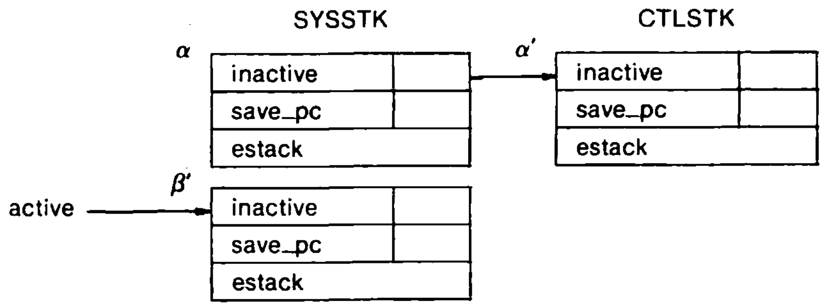
```
procedure drive()
   pop(SYSSTK)
   if \failure then {
      if \active.inactive then {
         push(SYSSTK, pop(CTLSTK))
         pc := top(SYSSTK).save_pc
         failure := &null
         }
      }
   else
      popto(CTLSTK, top(SYSSTK).inactive)
   active := top(SYSSTK)
end
```

Note that the value of pc after drive is different in the case of failure than it is when evaluation of the expression is successful.

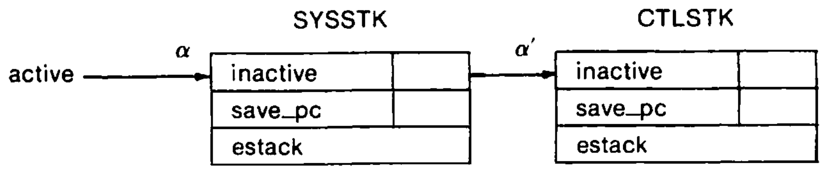
If SYSSTK and CTLSTK before the call to drive are



then if failure occurs, the stacks after the call are



If there is no failure, the stacks after the call are



Calls to the routines mark and drive enclose the code for each expression requiring an expression instance. A new expression instance is created by mark upon entry to the code for an expression and is destroyed by drive upon exit from the code for that expression.

Initially, failure has the null value, indicating that no failure has occurred. If a subexpression fails, failure is set to a non-null value and processing continues. After every subexpression that can conceivably fail, there is a test of the variable failure in the generated code. If this test detects failure, control branches immediately to the drive at the end of the expression.

In practice, this mechanism has proven to be unwieldy. Because the names of functions are global identifiers, the Icon translator can determine if a subexpression is capable of failing only when the subexpression contains only operators. In all other situations, it must be assumed that the subexpression might fail, and the code testing the failure condition must be inserted after all function and procedure calls. One improvement to this mechanism is to have operations directly perform the actions taken upon failure. This eliminates the need for the tests of failure after every subexpression and simplifies the translator. This enhancement is one of several presented in the following model for goal-directed evaluation, although it is not essential to the model.

## 4.2.2 The One-Stack Model of Goal-Directed Evaluation

Using two stacks as described above provides an effective implementation for goal-directed evaluation. Nevertheless, there are several disadvantages to using two distinct stacks. First, the use of a second stack complicates memory management for some machine architectures. Second, moving expression instances on to and off of the control stack involves additional overhead. It is possible to merge the control and system stacks into a single physical stack. The result is a more efficient implementation in both space and time.

The technique is to 'hide' inactive expressions instances *in place* on the system stack. Whenever a subexpression performs a save operation, a new expression instance is created containing a copy of the information necessary to continue processing (including the result being supplied from the subexpression). The new expression instance then becomes the active expression, and processing continues.

This approach has several advantages over the two-stack model. First, there is no need to copy an inactive instance back to the system stack when it is reactivated, since that instance is already on the system stack. Second, the amount of information that must be copied for an inactive instance is less than that required in the two-stack model.

As an example, consider evaluation of the expression

```
5 + (1 to 10) > x
```

Just before the to operation suspends, the active instance contains the value 5 as well as temporaries formed during evaluation of 1 to 10. (These temporaries are used to 'remember' the last result produced, the final result to produce, and the increment.) In the two-stack model, all of this information must be copied as part of the inactive expression instance. The one-stack model requires only copying the value 5 and the result produced by 1 to 10 into the new active instance, since that is all the information necessary to continue evaluation of the expression. The function copy_information copies the appropriate information from the currently active expression instance into the newly created expression instance.

The combination of the system and control stacks into a single stack is accomplished through a slight change in the expression marker: There is no longer any need for the inactive instance pointer. In those situations in the two-stack model where an active instance pointer points to an inactive instance, that inactive instance is now located immediately below the active instance. However, there is now a need for a passive instance pointer, since the next passive instance may not be the next instance on the stack.

Another change to expression instances is in the use of the activation address. Whereas the activation address in the two-stack model provides the point at which control is to resume in that instance, the activation address in the one-stack model provides the location at which control is to resume in the next expression on the stack when failure occurs in the current instance. While this change is mostly cosmetic, it simplifies the implementation of some control structures.

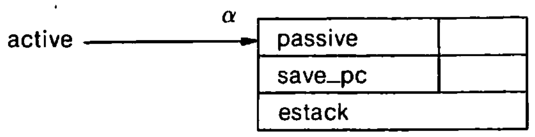
The estack field of expression instances is unchanged. Since expression stacks represent areas on the stack of expression instances, they do not affect the actual number of stacks needed in the implementation.

These changes necessitate some modifications to the routines that control goal-directed evaluation. In preparation for the presentation of additional language features, the global identifier active_stack is used to refer to the system stack.

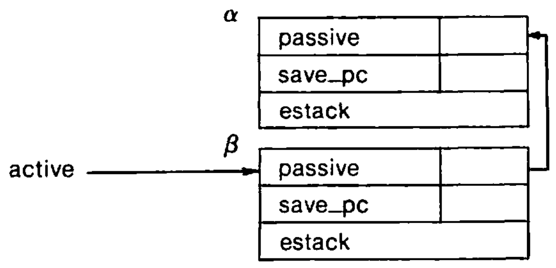
1. The procedure mark is the same as before, except that the passive instance pointer is set to point to the currently active instance.

```
procedure mark(failure_lab)
   push(active_stack, create_instance())
   top(active_stack).passive := active
   top(active_stack).save_pc := failure_lab
   active := top(active_stack)
end
```

   If the stack just before a call to mark is



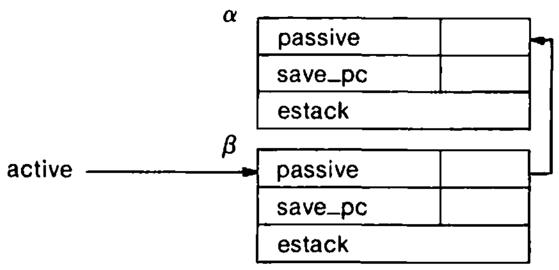
   then after the call the stack is



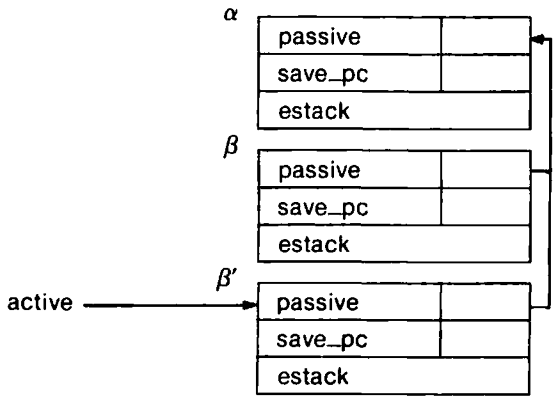
2. The procedure save hides the currently active instance in place on the stack when a subexpression suspends.

```
procedure save()
    push(active_stack, create_instance())
    copy_information(active, top(active_stack))
    top(active_stack).save_pc := pc
    active := top(active_stack)
end
```

If the stack just before a call to save is



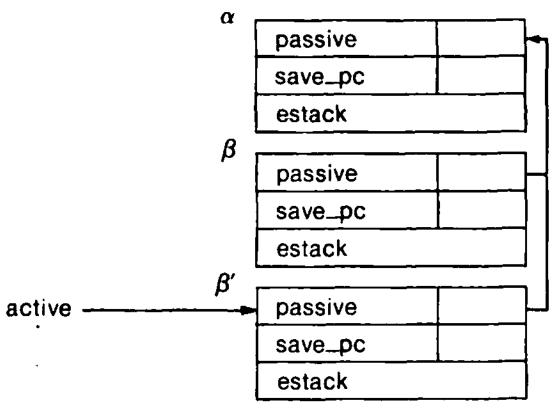
then after the call, the stack is



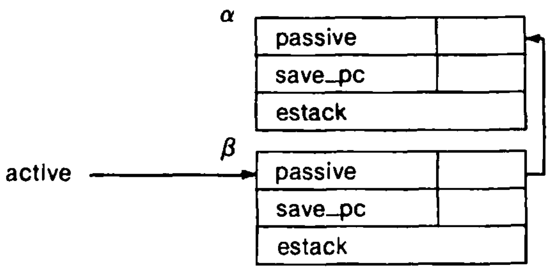
3. As stated earlier, subexpressions call the failure-handling mechanism directly when they fail. The routine failure handles all failures and is invoked by the subexpression that fails. It makes no difference when failure occurs whether or not there are any inactive instances for the currently active expression. The appropriate instance to reactivate is always the next instance on the stack, and the current save_pc is the location at which execution is to continue within that instance.

```
procedure failure()
    pc := active.save_pc
    pop(active_stack)
    active := top(active_stack)
end
```

If the stack before a call to failure is



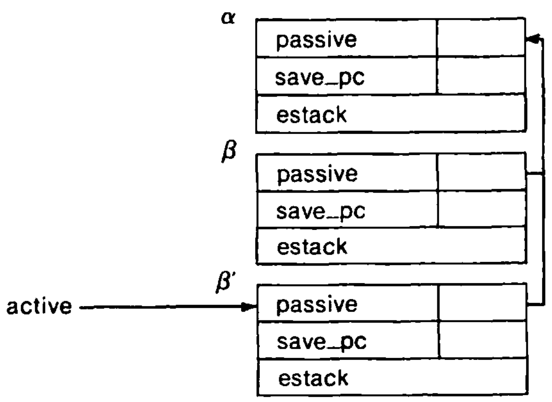
then after the call the stack is



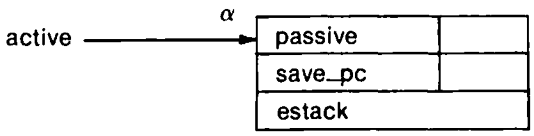
4. Because all expression failures are handled by failure, drive need only ensure that successful evaluation of an expression returns control to the next passive expression. The procedure drive is renamed unmark here. It does nothing more than pop the active stack to the passive expression instance.

```
procedure unmark()
   popto(active_stack, active.passive)
   active := top(active_stack)
end
```

If the stack before a call to unmark is



Then after unmark the stack is



### 4.2.3 Generated Code in the One-Stack Model

The code generation for a language using generators and goal-directed evaluation is straightforward. Because expression instances are created only at the points at which control decisions need to be made, most of the code production is identical to that in more conventional languages.

Control structures are another matter. The use of success or failure to control expression evaluation is directly reflected in the implementation of control structures. This section presents the generated code for some typical Icon control structures. A simple intermediate code, called *ucode*, is used to describe the code generated for these control structures.

The ucode instructions push, pop, goto, and invoke have conventional meanings. For simplicity, it is assumed that all operators, functions, and procedures are invoked through the same mechanism. For example, the ucode produced for the expressions

```
1 + 3
1 to 10
write(3)
```

is

```
push      1
push      3
invoke    +

push      1
push      10
invoke    to

push      3
invoke    write
```

In the code shown here, comments and annotations are enclosed in braces, and labels are terminated by colons, e.g.

```
lab1:          goto        lab1         {tight endless loop}
```

While operations that produce results do so by placing the result on the active expression stack, for convenience it is assumed that the location result also contains the result of the last operation.

The remaining ucode instructions deal exclusively with generators and goal-directed evaluation and correspond to the procedures described in the preceding section.

1. The instruction mark *lab* is the ucode form of the procedure mark(*lab*). The global label flab

is assumed to be the address of an invocation of failure(). Thus, mark flab propagates failure to the first passive expression instance upon failure of the marked expression instance.

2. The instruction unmark performs the same function as the procedure unmark().

3. The instruction fail corresponds to the procedure failure().

4. The procedure save() presented earlier is used within operators and functions to provide a result to the current active expression instance, and has no corresponding ucode representation. However, some control structures require that the active expression instance provide a result to the enclosing passive instance, without destroying any inactive instances of the currently active instance. The ucode instruction esave is used in these situations and corresponds to the procedure

```
procedure esave()
   pop(active_stack)
   push(active_stack, copy(active.passive))
   top(active_stack).save_pc := active.save_pc
   active := top(active_stack)
end
```

The approach is to replace the current active expression instance with a copy of the first passive expression instance (which then receives the result of the current active expression instance) and to continue processing.

### 4.2.4 The Generated Code for Control Regimes

If: one of the simplest control structures is

if *expr0* then *expr1* [else *expr2*]

If there is an else clause, the generated code is

```
                mark        lab1
                   {code for expr0}
                unmark
                   {code for expr1}
                goto        lab2
     lab1:
                   {code for expr2}
     lab2:
```

If *expr0* produces a result, the unmark pops the active stack to the first passive expression instance, and the then clause is evaluated in that instance. If *expr0* fails to produce a result, that same passive instance becomes the active instance. Control then branches via the failure mechanism to lab1 and the else clause is evaluated.

If the else clause is omitted and the control clause fails, the entire expression fails. Thus the code generated when the else clause is omitted is

```
        mark        flab
             {code for expr0}
        unmark
             {code for expr1}
```

*While*: the generated code for

while *expr0* do *expr1*

is

```
lab:
        mark        flab
             {code for expr0}
        unmark
        mark        lab
             {code for expr1}
        unmark
        goto        lab
```

Note that while-do itself does not produce a result.

*Repeat*: the generated code for

repeat *expr*

is

```
lab:
        mark        lab
             {code for expr}
        unmark
        goto        lab
```

In this case, if *expr* fails, control branches to the same point as when it succeeds.

*Every*: the generated code for

every *expr0* do *expr1*

is

```
        mark        flab
             {code for expr0}
        pop
        mark        flab
             {code for expr1}
        unmark
        fail
```

The pop after the control clause simply removes the result computed by that clause, since that result is ignored. Evaluation of the do clause takes place within a separate expression instance to limit evaluation to at most one result from that clause. A fail instead of an unmark occurs at the end of the code sequence and is evaluated in the expression instance for the control clause. This forces any inactive instances of the control clause to be reactived using save_pc as the reactivation address. Again, note that every-do itself does not produce a result.

Break and Next: the break and next control regimes are context sensitive operations that require special treatment during code generation. The break or next may appear in an expression instance that is nested within the instance in which the action is to take place. The code generator must know the depth of this nesting and issue enough unmarks to ensure that the operation of break or next occurs within the proper instance.

In the case of

```
break expr
```

the evaluation of *expr* occurs in the expression instance for the iteration control regime being exited. If *expr* is omitted, the outcome of break is failure. If the outcome of *expr* is success, program evaluation proceeds with the expression following the iteration control regime. On failure, goal-directed evaluation reactivates any generators in the active expression instance (the expression instance enclosing the evaluation of the iteration control regime). For example, in

```
every line := !&input do {
   if line == "stop" then
      break write("stop found")
   .
   .
   .
   }
```

the evaluation of break write("stop found") occurs within the expression instance for the do clause. Two unmarks are needed to make the instance enclosing the every the active instance in which write("stop found") is evaluated. The generated code for break write("stop found") in the example above is

```
unmark
unmark
   {code for write("stop found")}
goto        brklab
```

where *brklab* is the address of the expression following the every. If the argument to break is omitted, the generated code consists of the appropriate number of unmarks followed by fail.

The code generated by next is similar to that for break. In the case of all the iteration control regimes except every the generated code consists of the proper number of unmarks to make the instance enclosing the iteration control regime active. These unmarks are followed by a branch to the start of the code for the iteration control regime. If the next occurs in the do clause of an every expression, one fewer unmark is generated so that the expression instance for the first argument to every becomes active. If the next occurs in the first argument of every, no unmarks are generated. In either case, a branch to flab is then generated to activate any inactive instances in the first argument.

**Not**: the generated code for

```
not expr
```

is

```
                mark        lab
                   {code for expr}
                unmark
                fail
lab:
                push        &null
```

The generated code for

```
expr1 | expr2
```

is

```
                mark        lab1
                   {code for expr1}
                esave
                push        result
                goto        lab2
lab1:
                   {code for expr2}
lab2:
```

The instruction esave is used to make the expression instance for evaluating the left control expression inactive, so that failure in the surrounding expression instance reactivates the left control expression before attempting evaluation of the right control expression.

**Limit**: the generated code for

```
expr1 \ expr2
```

is more complicated and requires the introduction of two new ucode instructions, limit and lsave.

The procedural form of limit is

```
procedure limit()
  if result <= 0 then failure()
end
```

The procedure limit checks the current result and succeeds if the result is positive. The procedure limit leaves this result upon the stack to function as a counter of results left to produce from *expr1*.

The instruction lsave is responsible for maintaining the count of results produced. The top of the stack of temporaries for the first passive expression instance is the count of results left to produce.

If the last result is being produced, then lsave is similiar to unmark. If it is not the last result, then lsave performs the same function as esave.

The procedural form of lsave is

```
procedure lsave()
   top(active.passive.estack) -:= 1
   if top(active.passive.estack) > 0 then
      esave()
   else
      unmark()
end
```

The generated code for

*expr1* \ *expr2*

is thus

```
       {code for expr2}
limit
mark          flab
       {code for expr1}
lsave
pop
push          result
```

The last two instructions replace the count of remaining results with the result of *expr1*.

Reval: as a final example of generated code, consider

|*expr*

The difficulty in implementing repeated evaluation arises when the control expression fails to produce any result, in which case repeated evaluation fails rather than attempting to evaluate the control expression anew. If this condition were removed, the generated code would be

```
lab:
                mark          lab
                       {code for expr}
                esave
                push          result
```

With the failure condition, the code is

```
lab:
                    mark       flab
                        {code for expr}
                    chfail     lab
                    esave
                    push       result
```

The ucode instruction chfail changes the activation address from flab to lab after the control expression has produced a result. Thus if no result is produced by the control expression, the failure is propagated to the first passive instance enclosing the repeated evaluation. If at least one result is produced, chfail insures that subsequent failure causes the expression to be evaluated anew.

To force re-evaluation of the expression, chfail changes the activation address of the expression instance immediately following the passive instance awaiting a result from the repeated evaluation. This activation address is the point at which evaluation continues in the passive instance when failure occurs in the repeated evaluation. An additional primitive operation is used to gain access to that activation address. The function one_above(i) returns a pointer to the expression instance containing the activation address that is used when reactivating expression instance i.

The procedural form of chfail is

```
procedure chfail(failure_label)
   one_above(active.passive).save_pc := failure_label
end
```

## 4.3 Co-Expressions

Co-expressions can be added to languages that do not include generators or goal-directed evaluation. Much of the expressiveness of co-expressions is lost in such a situation, however, reducing co-expressions to a conventional coroutine facility. When integrating co-expressions into a language that includes generators, the ease of implementing co-expressions depends to a large degree on the implementation chosen for generators.

Forming a co-expression from an expression involves the creation of a *co-expression instance* that encapsulates the the evaluation of the expression. While co-expressions can be implemented using either the one- or two-stack model of goal-directed evaluation, implementation using the one-stack model is simplest, and it is used here.

A co-expression instance encapsulating an expression can be viewed as the stack used to maintain any expression instances created during evaluation of the expression and a location counter for that expression. It is convenient to treat the location counter and stack in which program execution is initiated as a co-expression instance. The expression in which evaluation of an Icon program is initiated consists of an invocation of the procedure main. The co-expression instance in which program evaluation is currently taking place is termed the *current* co-expression, and its stack is termed the *active stack*. Thus each co-expression maintains its own version of the system stack.

The current co-expression is pointed to by the global variable current. Evaluation, or *activation*, of a co-expression is a straightforward process of switching current from one co-expression instance to another. The first co-expression instance is termed the *activator* of the second. The expression instance on top of the stack for the activator becomes a passive instance, awaiting a result from the activated co-expression. The instance on top of the activated co-expression stack becomes the active expression instance.

When the activated co-expression produces a result, that result is transmitted back to the activator, where processing continues.

The ucode instruction create *lab* produces a co-expression instance for evaluating the co-expression whose code begins at *lab*. This co-expression instance is represented as

| | |
|---|---|
| activator | |
| pc | |
| sstack | |

and is modeled as a record

```
record coexpr(activator, pc, sstack)
```

where activator points to the current activator of that co-expression, pc is the location counter for that instance, and sstack is the system stack for expressions instances formed during evaluation of the co-expression.

The instruction create has the procedural form

```
procedure create_coexpr(first_instr)
   push(active.estack, coexpr(&null, first_instr, stack()))
   push(top(active.estack).sstack, create_instance())
end
```

Note that an initial expression instance is built into the co-expression instance stack. This is done so that the activation process need not determine whether or not the co-expression instance stack is empty.

The procedure activate switches to a new co-expression instance. Any co-expression that is the activator of some other co-expression has a passive expression instance on the top of its stack. Since passive instances are waiting for a result to be provided from some other instance, it is reasonable to transmit a result to the new co-expression instance. The procedure activate provides the current result to the new co-expression instance. If the activated co-expression is not the activator of some other co-expression, the result is ignored.

```
procedure activate(coexpr)
   coexpr.activator := current
   current.pc := pc
   current := coexpr
   active_stack := current.sstack
   active := top(active_stack)
   pc := current.pc
   push(active.estack, result)
end
```

The procedure coreturn provides a result from one co-expression to its activator.

```
procedure coreturn()
    current.pc := pc
    current := current.activator
    active_stack := current.sstack
    active := top(active_stack)
    pc := current.pc
    push(active.estack, result)
end
```

There are few differences between activate and coreturn. The procedure activate sets the activator field of the activated co-expression, while coreturn simply returns control to its activator.

If a co-expression becomes exhausted, failure is reported to the activator. The procedure cofail is invoked when a co-expression fails.

```
procedure cofail()
    current.pc := pc
    current := current.activator
    active_stack := current.sstack
    active := top(active_stack)
    failure()
end
```

## 4.3.1 The Generated Code for Co-Expressions

The ucode operations create, coreturn, and cofail correspond to the procedural forms given earlier. Activation,

*expr1* @ *expr2*

is like any other binary operator, and has generated code

```
        {code for expr1}
        {code for expr2}
invoke        activate
```

The generated code for

create *expr*

is a bit more complex than that generated for co-expression activation. The approach is to branch around the code generated for the co-expression and do a create with a pointer to the code for the co-expression. Hence the Icon expression

create *expr*

produces the ucode instructions

```
                goto      lab2
lab1:
                pop
                mark      clab
                   {code for expr}
                coreturn
                fail
lab2:
                create    lab1
```

The pop at the beginning of the co-expression code removes the result provided by activate, since that result is ignored when the co-expression is first activated.

When a co-expression is exhausted, it fails any time that it is subsequently activated. The mark clab causes a branch to the universal label clab when the co-expression is exhausted. The code at clab is

```
clab:
                cofail
                goto      clab
```

which repeatedly transmits failure back to any activator of the co-expression.

Finally, the fail after coreturn forces the co-expression to produce its next result the next time it is activated.

## 4.4 Implementation Details

For pedagogical purposes, a ucode generator and a ucode interpreter for a small subset of Icon have been written using the above routines and are contained in Appendices A and B, respectively. The use of Icon obscures a number of practical considerations that are encountered when implementation is attempted using a conventional system implementation language, however. A major difficulty arises because system stacks are typically addressed in terms of machine words or bytes, not expression instances as in the Icon model.

This section describes modifications to the models that are necessary to add goal-directed evaluation and co-expressions to a language based upon conventional implementation techniques. The modified models reflect the general approaches taken in the implementation of Cg and Icon, though both Cg and Icon include features and optimizations not presented here. Appendix C contains a ucode interpreter based upon the modified one-stack model.

### 4.4.1 Goal-Directed Evaluation

Because conventional implementation languages treat the system stack as a stack of words or bytes, expression instances are represented by expression markers separating expression stack areas. In the one-stack model, it is assumed that active and any passive instance pointers point to expression markers. Expression markers are of some fixed length, while expression stack areas vary in size depending upon the number of temporary results created when each instance is active. The *modified* one-stack model assumes this more conventional layout of system stacks.

In a conventional system stack, the top of the stack is pointed to by the *stack pointer*. In the modified one-stack model, this stack pointer corresponds to a pointer to the top of the expression stack for the active expression instance, and is represented by the global identifier sp.

When expression instances are popped from the system stack, two actions occur. First, active is changed to point to the expression marker for the new active instance. Second, sp is changed to point to the top of the new active instance.

When there are no inactive instances of the current active expression, these two operations are accomplished by setting sp to the current value of active, and setting active to the current passive instance pointer. These actions are sufficient regardless of whether or not the active instance is producing a result. They are not sufficient, however, when there are inactive instances of the current active expression.

There are two cases to consider when there are inactive instances of the current expression.

1. If the active expression has failed to produce a result, then active is changed to point to the next expression marker on the stack, and sp is changed to point to the top of the expression stack area for this new active instance.

2. If the active expression has succeeded in producing a result for some passive instance, then active is changed to point to the expression marker for that passive instance, and sp is changed to point to the top of the expression stack area for that passive instance.

Accomplishing the proper operation in both cases requires that expression markers include two supplemental pointers. First, an inactive instance pointer is needed for resetting active during reactivation. Second, a *saved stack pointer* is needed for resetting sp to the top of the expression stack area in the enclosing passive instance.

An expression instance in the modified one-stack model has the form

| | |
|---|---|
| passive | |
| inactive | |
| save_sp | |
| save_pc | |
| estack | |

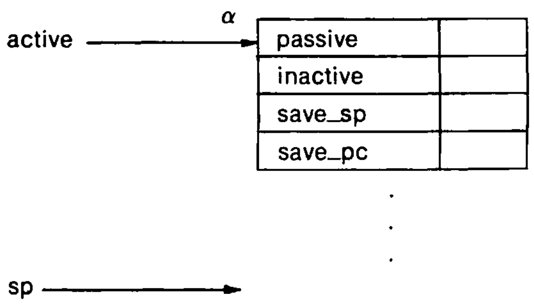
with the first four fields constituting the expression marker.

Given these changes, the primitive operations on expression instances can be rewritten for a conventional system stack. The assumptions are that the system stack is addressed on a word basis and that all pointers into the stack are negative offsets from the base of the stack. Hence push decrements sp and pop increments sp.

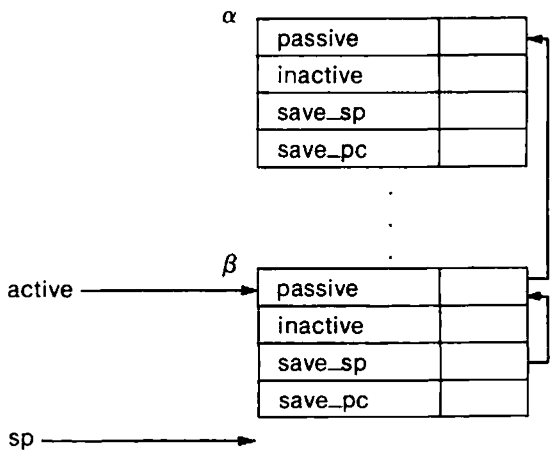
1. The procedure mark pushes a new expression marker onto the stack.

```
procedure mark(failure_lab)
local sactive
   sactive := sp
   push(active_stack, active)
   push(active_stack, &null)
   push(active_stack, sactive)
   push(active_stack, failure_lab)
   active := sactive
end
```

If the stack before a call to mark is



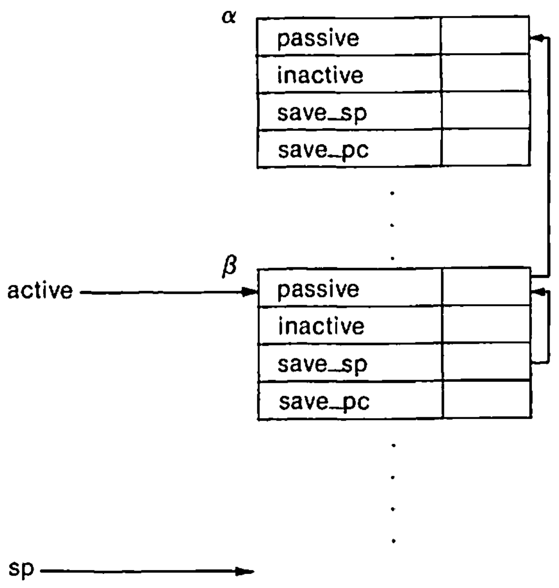
after the call the stack is



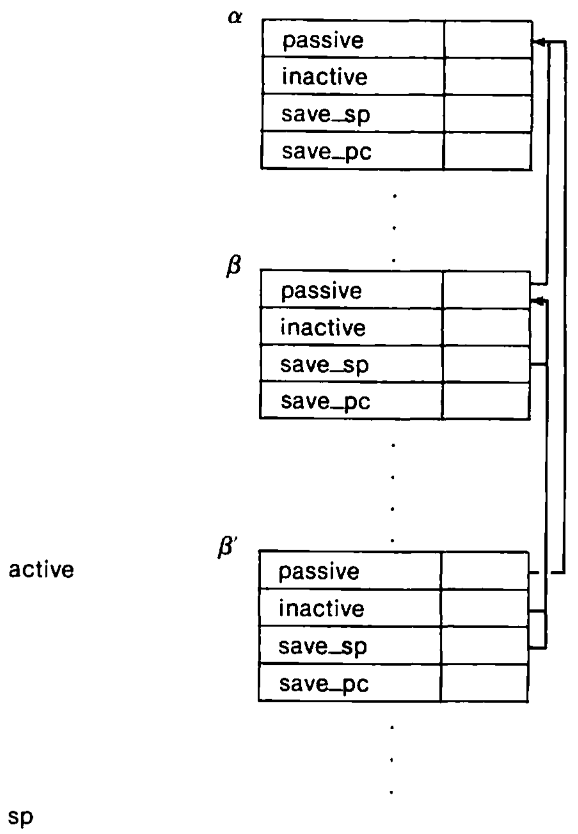
2. The procedure save must update inactive and save_sp as it "hides" the current active instance. Note that the routine copy_information is replaced by a simple every loop.

```
procedure save()
local sactive
   sactive := sp
   push(active_stack, active_stack[active])
   push(active_stack, active)
   push(active_stack, active_stack[active-2])
   push(active_stack, pc)
   every
      push(active_stack, active_stack[active-4 to sactive+1 by -1])
   active := sactive
end
```

If the stack before a call to save is



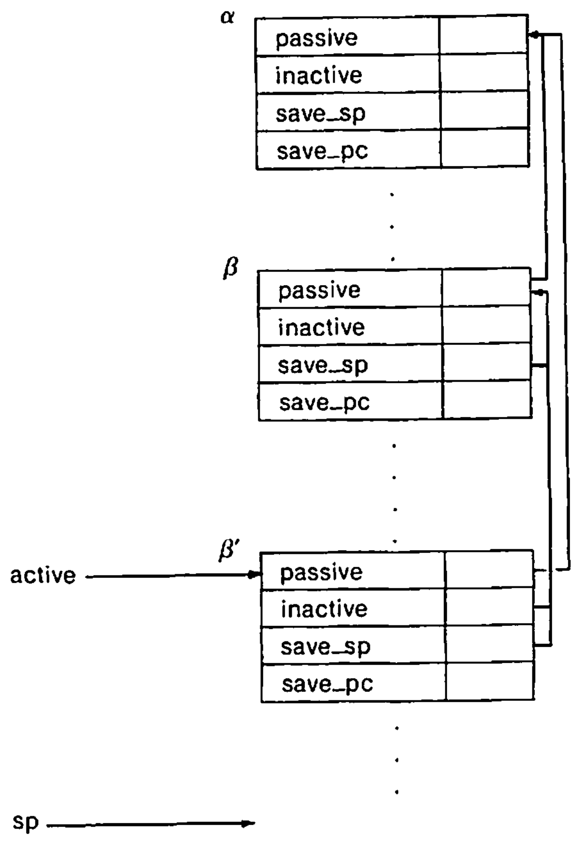
then after the call the stack is



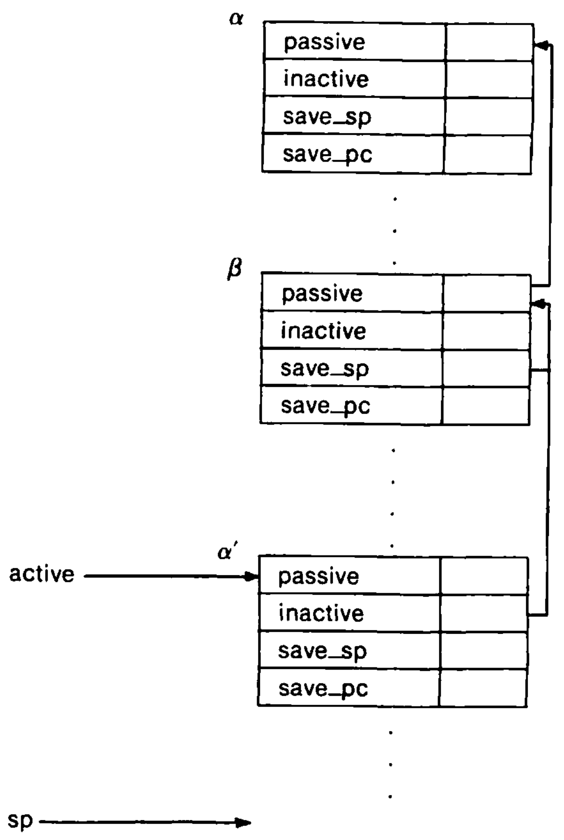
3. The changes to esave are similar to those required by save. esave must ensure that inactive points to the next expression marker on the stack.

```
procedure esave()
local passive, inactive, ssp, spc
   passive := active_stack[active]
   inactive := active_stack[active-1]
   ssp := active_stack[active-2]
   spc := active_stack[active-3]
   sp := active
   push(active_stack, active_stack[passive])
   push(active_stack, \inactive|passive)
   push(active_stack, active_stack[passive-2])
   push(active_stack, spc)
   every
      push(active_stack, active_stack[passive-4 to ssp+1 by -1])
end
```

If the stack prior to a call of esave is



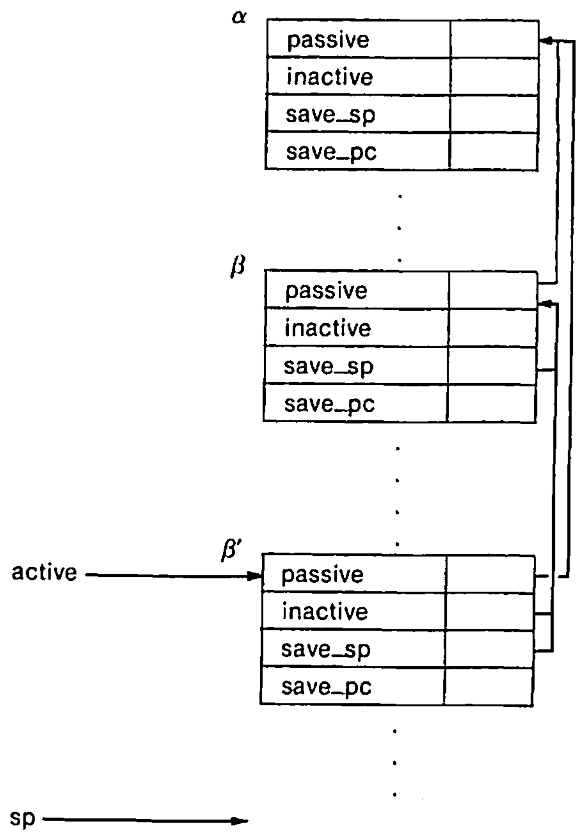
then after the call it is



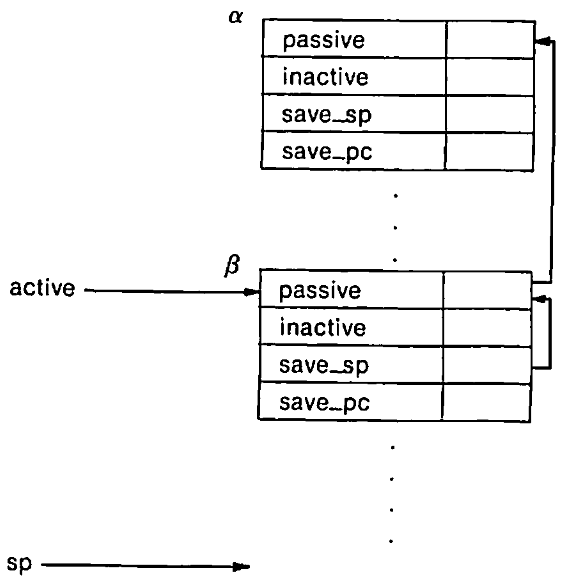
4. The procedure failure determines whether there is an inactive instance to reactivate. If there one, it is reactivated. Otherwise, failure reactivates the enclosing passive instance and sets pc to the value of save_pc for that passive instance.

```
procedure failure()
   pc := active_stack[active-3]
   sp := active
   if \active_stack[active-1] then
      active := active_stack[active-1]
   else
      active := active_stack[active]
end
```

If the stack before a call to failure is



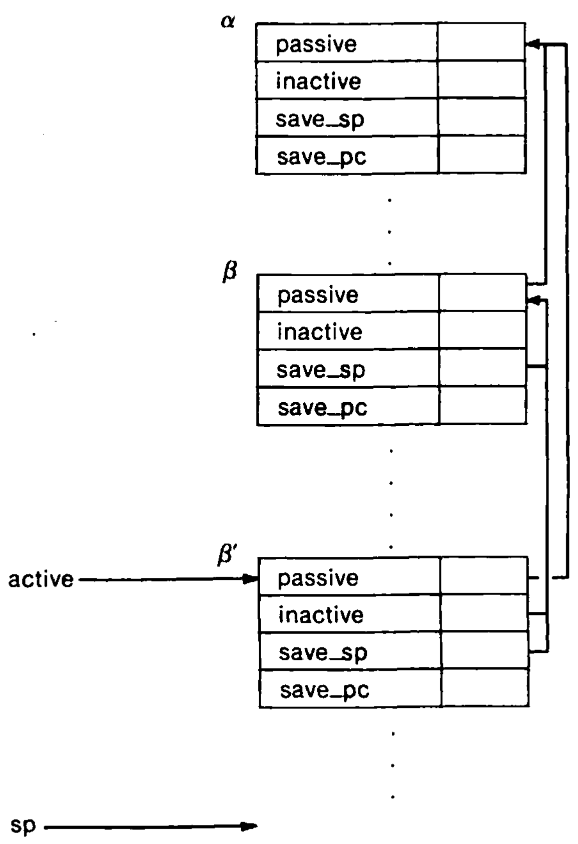
then after the call the stack is



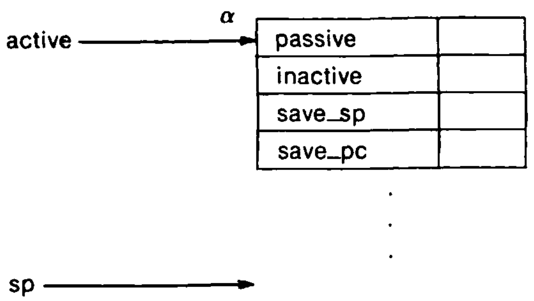
5. The procedure unmark pops all inactive instances of the active instance from the stack.

```
procedure unmark()
   sp := active_stack[active-2]
   active := active_stack[active]
end
```

If the stack before a call to unmark is



then after the call the stack is



6. The procedure lsave works as in the original one-stack model.

```
procedure lsave()
local top_passive
   top_passive := active_stack[active-2]
   if (active_stack[top_passive+1] -:= 1) > 0 then
      esave()
   else
      unmark()
end
```

7. Finally, chfail is able to access the appropriate activation address directly.

```
procedure chfail(failure_lab)
local one_above
   one_above := active_stack[active-2]
   active_stack[one_above-3] := failure_lab
end
```

These are all the changes needed to implement the one-stack model of goal-directed evaluation using a conventional system stack. However, interfacing goal-directed evaluation with other language features may require additional modifications. Storage reclamation is the most notable example in Icon. The storage reclamation algorithm must locate all valid data items. To do so requires that the system stack be *tended* (searched for valid data) (Hanson 1977).

In the modified one-stack model, all expression stack areas contain valid data and must be tended. This is not difficult in itself; the pointers active and sp as well as the pointers in the expression markers are sufficient to locate all the expression stack areas. The problem is that not all of the information within an expression stack area is necessarily valid Icon data. Inactive instances may contain information left by run-time support routines, which must be skipped over during tending.

Fortunately, the information to be ignored is always at the top of the expression stack area, and an additional pointer can be associated with inactive instances to give the separation point between valid Icon data and information left by any run-time support routines. This pointer is called the expression area *boundary* (Coutant and Wampler 1981).

Besides assisting in the storage reclamation process, the boundary helps distinguish functions and operators from user defined procedures. An inactive instance with information above its boundary is an instance for evaluating a suspended function or operator. An instance with no information above its boundary is an instance for the evaluation of a procedure. This information is also useful in Icon's tracing mechanism, which traces procedure reactivation, but not function or operator reactivation.

### 4.4.2 Co-Expressions

Co-expressions can be implemented as described previously, with the exception that active and sp must be preserved with each co-expression. The simplest solution is to push active and sp onto the co-expression stack each time that co-expression activates some other co-expression and get the new active and sp from the top of the activated co-expression. A particular machine architecture may cause severe problems with the implementation of co-expressions; an example is the PDP-11/70. The PDP-11 does not have stack-based addressing for stack operations. Rather, pointers into the stack reference absolute memory locations within the user's data region. This makes relocation of stacks during Icon's storage reclamation process difficult, as all pointers into each stack must be tended.

Tending the pointers within the expression markers is possible, since they are known to be pointers into the stack. However, inactive instances may contain information above their boundary, and this information may contain pointers into the stack that are unknown to the storage reclamation process. It is therefore impossible to relocate co-expression stacks. The problem of identifying unknown pointers also makes it impossible to copy a co-expression with the Icon function copy.

In the implementation of co-expressions used in Icon, a fixed-sized space is allocated for each created co-expression to serve as the co-expression stack. This space is never relocated during the storage reclamation process, but is tended.

A final difficulty with co-expressions arises from their use as data objects. As a data object, the lifetime of a co-expression may exceed the lifetime of the procedure in which it is created. Variables that are local to the procedure and that are referenced within the co-expression must exist as long as the co-expression exists. A co-expression is provided copies of all the current local variables when that co-expression is created. These copies are maintained with the co-expression, freeing the scope of the co-expression from the scope of the creating procedure and eliminating any problem similar to the FUNARG problem in LISP (Moses 1970).

## 4.5 Performance of the Implementation

The performance of the implementations given here is difficult to measure. Programs written using goal-directed evaluation or co-expressions differ greatly in style and approach from similar programs written without these language features. A few observations are possible, however.

### 4.5.1 Goal-Directed Evaluation

In situations in which there are no inactive instances, the system stack differs little in appearance from the system stacks for conventional stack-based languages. Only a few extra words (the expression marker) are added to separate expression instances. The number of expression markers is reduced because expression instances are only needed at points of program flow control. Some of the information within the expression marker is needed only for inactive instances, and can be removed from other expression markers, reducing the number of words per expression marker.

Finally, both mark and unmark are simple operations, and can be implemented with a few machine instructions. Thus the impact on the efficiency of other language features is slight, especially in light of the expressiveness added by the language features implemented with these operations.

It is when an expression instance becomes inactive that the two major sources of inefficiency in the performance of generators occur. First, there is the overhead involved in hiding that instance on the stack. This overhead is reduced slightly by only copying the required portion of the expression instance.

Second, an instance cannot suspend with a variable pointing to information contained within the expression stack area for that expression, since that area may not exist by the time the variable is referenced. Such a variable must be dereferenced when the instance suspends. Since the same situation occurs when a procedure returns a result, this problem is not endemic to generators.

Unmarking, reactivating, and propagating failure are all efficient operations amounting to little more than resetting the system stack pointer to the appropriate place. Note that reactivation in the one-stack model is thus considerably more efficient than reactivation in the two-stack model, which must copy the reactivated instance from the control stack back onto the system stack.

**4.5.2 Co-Expressions**

Co-expression creation and refreshing are fairly expensive, though relatively infrequent, operations. Space for the co-expression stack must be allocated and the local variables for the current procedure must be copied. However, activation of a co-expression is a simple operation, accomplished in a few machine instructions. As with goal-directed evaluation, the major source of inefficiency with co-expression activation is that variables pointing to values within the activating co-expression must be dereferenced.

The impact that co-expressions have on other language features depends in part upon the sophistication of the underlying machine architecture. The source of the impact is in detecting stack overflow of the co-expression stack.

Again, the PDP-11/70 provides a case in point. The hardware of the 11/70 provides stack overflow checking for the primary system stack by detecting when a push operation causes the stack pointer to cross into the user's data region. However, the stack spaces for additional co-expressions lie entirely within the user's data region, and the hardware does not detect overflow on these stacks. Adequate stack overflow detection of co-expression stacks on the 11/70 requires that software checks be inserted into the code. Since it cannot be determined whether a section of code will be executed in the main system stack or another co-expression stack, these checks must be inserted throughout the code, degrading the performance of all language features. This is not a problem on machines such as the DEC-10, which permit the specification of an upper bound for each stack.

# Chapter 5

# Conclusions

## 5.1 Generating Expressions and Co-Expressions

Generating expressions are a powerful programming facility because they add expressiveness to a programming language. The ability to produce a sequence of results during the evaluation of a single expression instance increases the *information density* of expressions. Increasing the amount of information that can be described in an expression provides a more concise representation for a wide variety of algorithms. This permits programmers to concentrate more on the development of an algorithm, and less on the details of its implemention.

In early versions of Icon, the capabilities of generating expressions were not well understood. Most control regimes for the evaluation of generators were patterned after the control regimes in more conventional languages. The only control regimes that were designed entirely upon the operation of generators were **Alternation**, **Every**, and **Goal**. **Alternation** permits the programmer to form new generators at the expression level by concatenating result sequences to form new result sequences. **Every** permits programmers to explicitly process the entire result sequence of a generating expression. **Goal** directs the focus of attention in an expression toward the final result, leaving much of the computational detail necessary to attain that result to the operation of **Goal**.

The concept of result sequences has led to better understanding of generators. This, in turn, has resulted in the development of additional control regimes that provide increased flexibility in the manipulation of generating expressions. Control regimes such as **Limit** and **Reval** extend the programmer's ability to form new generators at the expression level by providing additional methods for the construction of new result sequences.

Result sequences can be used to describe the operation of control regimes found in other languages. For example, as an extension to SNOBOL4 pattern-matching, Doyle introduces the concept of *forward alternation* to avoid unnecessary processing that occurs in many uses of ordinary alternation (Doyle 1975). Forward alternation is similar to ordinary alternation, in that alternatives are evaluated until one succeeds. Once an alternative of forward alternation succeeds, however, the remaining alternatives are discarded. Whereas Doyle had to introduce another operator to achieve this, forward alternation is merely an instance of limiting the result sequence of a generating expression.

Forward alternation, with Doyle's syntax

$$e_1 \,!\, e_2$$

can be represented as the $\mathfrak{L}$-expression

$$(e_1 \mid e_2)\backslash 1$$

and the equivalent &-expression

> **Limit**: [**Alternation**: $e_1$,$e_2$] , 1

That is, forward alternation simply limits the result sequence of **Alternation** to at most one result. The use of result sequences to describe forward alternation is not only succinct, it is more precise than the explanation given by Doyle. Forward alternation is one example of *fastback* backtracking (Allison 1978). Fastback backtracking simply limits the result sequence of an expression to at most one result. As shown above, **Limit** provides a generalized form of fastback backtracking.

Co-expressions represent a significant step in increasing the functionality of generating expressions, since they free the evaluation of generating expressions from their lexical sites and permit access to the elements of the result sequence for a generating expression when and where needed. In this sense, co-expressions represent the instantiation of result sequences into the programming language as data objects that can be manipulated in much the same way as other data objects are manipulated. Expressions can be formed that interleave elements of two result sequences, or to access subsequences of result sequences.

Co-expressions also provide insight into the operation of coroutine facilities. Most languages that incorporate coroutines do so by associating the coroutine mechanism with procedures. In actuality, it is the expression instance containing the *invocation* of a procedure that functions as the coroutine. Co-expressions make this clear by associating the coroutine mechanism with expression instances, rather than with procedures.

The concept of expression instances is also a useful descriptive tool leading to a better understanding of the operation of goal-directed evaluation and co-expressions. Considering expression evaluation as occurring within expression instances and then examining goal-directed evaluation and co-expressions as operations upon expression instances simplifies the development and analysis of implementation techniques for these language features.

## 5.2 Future Research

The development of new control regimes for the evaluation of generating expressions is a research area that is still relatively unexplored. At this point, there are only a handful of control regimes for composing generators at the expression level (**Alternation**, **Limit**, and **Reval**), and only two language features for constructing generators at the procedure level (suspend and fail).

Besides developing new control regimes, extensions to the current control regimes should be examined. Goal-directed evaluation is such a powerful programming feature that alternatives and extensions to this control regime might prove useful. For example, one of the difficulties in controlling generators with goal-directed evaluation is that there is no mechanism for limiting the result sequences for individual operations without also limiting the result sequences for arguments to that operation. Consider the expression

```
every find("icon", line := !&input) do
   write(line)
```

which outputs any input line containing the substring icon. However, if an input line contains more than one occurrence of icon, that line is output more than one time. The problem is that the result sequence for find (s1, s2) contains the locations of all occurrences of s1 in s2.

Attempting to limit find to at most one result in the above example also limits !&input to at most one result. Thus,

```
every find("icon", line := !&input) \ 1 do
   write(line)
```

outputs only the first line containing icon. One solution to this problem is to split the expression into its component parts as in

```
every line := !&input do
   if find("icon", line) then
      write(line)
```

Another solution might involve the use of an alternative form of function invocation that permits limitation of specific operations, without limiting the arguments. Let $\Upsilon$ denote this alternative function invocation mechanism, with the following relationship to $\Gamma$:

$$\Upsilon{:}\mathit{function} = \mathfrak{Limit}{:}[\,\Gamma{:}\mathit{function}\,], 1$$

That is, $\Upsilon$ produces at most one result from the invocation of a function.

$\Upsilon$ may be applied during goal-directed evaluation in place of $\Gamma$ to limit a particular operation without limiting the arguments to that operation. The operation of $\Upsilon$ is an excellent example of how goal-directed evaluation can be extended, and provides insight into several aspects of goal-directed evaluation that must be considered in any extension.

The application of $\Gamma$, as with the use of goal-directed evaluation, is implicit in $\mathfrak{L}$-syntax. Allowing the specification of multiple function invocation mechanisms means that some representation must be used to make the mechanism explicit in $\mathfrak{L}$-syntax. For example, an operation that is to be invoked with $\Upsilon$ instead of $\Gamma$ might be tagged in $\mathfrak{L}$-syntax with $\triangledown$. For example, the following $\mathfrak{L}$-expression would output every input line that contains the string icon, with no line being output more than once.

```
every ▽find("icon", line := !&input) do
   write(line)
```

As a second example of an application of $\Upsilon$, the following code segment converts strings of the form *hh-mm-ss* into seconds.

```
secs := 0
scan s using
   every secs := ▽tab(upto('-')|0) + 60 * secs do
      move(1)
```

Here the invocation of tab is performed by $\Upsilon$, thereby limiting tab without limiting the invocation of upto. Without $\Upsilon$, this code segment must be rewritten as

```
secs := 0
scan s using
   every secs := xtab(upto('-')|0) + 60 * secs do
      move(1)
```

where the procedure xtab is used to isolate the invocation of tab so that tab can be limited to one result without limiting upto. The procedure xtab can be written

```
procedure xtab(pos)
   return tab(pos)
end
```

Other alternatives to the current goal-directed evaluation control regime might include using a queue in the implementation to hold inactive instances instead of a stack. The use of a queue would not affect the size of the result sequence for an expression but it would affect the order in which results are generated if the expression contains multiple generating subexpressions. The single-stack implementation presented here could not be used to implement a queue-based goal-directed evaluation scheme, although a more general single-stack model could be used (Bobrow and Wegbreit 1973).

Another area for additional research is result sequences. Result sequences have already suggested a number of control structures for generators, as well as providing insight into the relationship between Icon's expression evaluation mechanism and those of other languages. Result sequences represent the first attempt at introducing a useful formal semantics for generators and goal-directed evaluation. Result sequences are too specialized to provide a general formal semantics for these language features, however, since they describe only some of the static aspects of expression evaluation.

One area in which research has already begun is the incorporation of generators and goal-directed evaluation into other languages. The language Cg shows that generators can be a useful addition to more conventional languages (Budd 1981). However, more work is needed to determine both the impact and the limitations involved when goal-directed evaluation and generators are integrated with the features of other languages.

Finally, co-expressions are a fertile ground for research. For instance, the control regimes for co-expressions are currently based upon the control regimes for generators, and are occasionally ill-suited to co-expression analysis. As an example, the procedure compare presented in Chapter 3 works only if the result sequences for its arguments are the same length. Using the current control regimes, a more general solution is not straightforward, with or without co-expressions. However, if there is a control regime that guarantees the same number of activations are attempted on two (or more) co-expressions, then there is a straightforward solution.

A *conjunction control regime* with proposed $\mathfrak{L}$-syntax

*expr1* and *expr2*

could evaluate both *expr1* and *expr2*, failing if evaluation of either fails. Unlike the conjunction operator, and would evaluate *expr2* even if *expr1* fails. The conjunction control regime could then be used in a more general version of compare:

```
procedure compare(cx1, cx2)
local r1, r2
   while r1 := @cx1 and r2 := @cx2 do
      if r1 ~=== r2 then fail
   if *cx1 ~= *cx2 then fail
   return
end
```

In this version, a simple comparision of the sizes of cx1 and cx2 is used to determine if their result sequences are the same length. This test is sufficient since it is known that the same number of activations were attempted on both co-expressions.

## Appendix A — A Translator for a Subset of Icon

The following program is a sample translator for a subset of Icon. The program reads an Icon expression and produces the corresponding ucode. A recursive descent parser recognizes valid expressions from a representative subset of Icon. The parser does not handle semicolon insertion and treats the input as a compound expression. Once the expression is parsed successfully, the ucode is generated. The program runs under Version 4 of Icon.

```
## main(args) - parse an Icon expression and generate ucode

procedure main()
   gencode(parse(getline()))
end

## getline() - gets source lines.
#    Essentially, standard input is treated as a compound
#    statement (getline supplies opening and closing braces).

procedure getline()
local line
   line := "{"
   every line ||:= " " || !&input
   return line || " }"
end

## parse tree node declarations

record Binop(opcode, left, right)
record Unop(opcode, e)
record Noop(opcode, e)
record Break(e)
record Next()
record Compound(left, right)
record Create(e)
record Every(ctl, docls)
record Ident(var)
record If(ctl, tcls, ecls)
record Not(e)
record Or(left, right)
record Limit(left, right)
record Reval(e)
record While(ctl, docls)
```

```
## parse(line) - root of recursive descent parser
#     Parse only recognizes a limited subset of
#     Icon.

procedure parse(line)
   scan line using
      return expr()
end

procedure expr()
local left
   left := expr1()
   sb()
   ="&" & return Binop("&",left,expr())
   return left
end

procedure expr1()
local left
   sb()
   ="create " & return Create(expr1())
   left := expr2()
   sb()
   =":=" & return Binop(":=",left,expr1())
   return left
end

procedure expr2()
local left
   left := expr3()
   sb()
   while ="\\" & left := Limit(left,expr3())
   return left
end

procedure expr3()
local left
   left := expr4()
   sb()
   while ="to " & left := Binop("to",left,expr4())
   return left
end

procedure expr4()
local left
   left := expr5()
   sb()
   ="|" & return Or(left,expr4())
   return left
end
```

```
procedure expr5()
local left
   left := expr6()
   sb()
   while left := Binop(=("="|"~="|"<="|"<"|">="|">"),
                       left,expr6())
   return left
end

procedure expr6()
local left
   left := expr7()
   while left := (sb() & Binop(=("+"|"-"),left,expr7()))
   return left
end

procedure expr7()
local left
   left := expr8()
   while left := (sb() & Binop(=("*"|"/"|"%"),left,expr8()))
   return left
end

procedure expr8()
local left
   left := expr9()
   while left := (sb() & Binop(="@",left,expr9()))
   return left
end

procedure expr9()
   sb()
   ="not" & return Not(expr9())
   ="|" & return Reval(expr9())
   ="@" & return Binop("@","",expr9())
   return Unop(="!", expr9()) | expr10()
end
```

```
procedure expr10()
local e, e1
static alpha, digits
   initial {
      digits := '0123456789.r'
      alpha := &ucase ++ &lcase ++ digits
      }
   sb()
   ="if " & return expr11()
   ="while " & return expr12(While)
   ="every " & return expr12(Every)
   ="break" & return Break(expr() | &null)
   ="next" & return Next()
   ="write(" & e := expr() & sb() & =")" & return Unop("write", e)
   ="writes(" & e := expr() & sb() & =")" & return Unop("writes", e)
   ="read()" & return Noop("read")
   ="\"" & e := tab(upto("\"")) & move(1) & return e
   ="{" & e := expr13() & sb() & ="}" & return e
   ="(" & e := expr() & sb() & =")" & return e
   return numeric(tab(many(digits))) | identifier(tab(many(alpha)))
end

procedure expr11()
local ctl, tcls, ecls
   ctl := expr()
   sb()
   ="then " & {
      tcls := expr()
      sb()
      ="else " & ecls := expr()
      return If(ctl,tcls,ecls)
      }
   stop("Error in <if> expression")
end

procedure expr12(pnode)
local ctl, docls
   ctl := expr()
   sb()
   ="do " & docls := expr()
   return pnode(ctl,docls)
end

procedure expr13()
local e
   e := expr()
   sb()
   =";" & return Compound(e,expr14())
   return e
end
```

```
procedure identifier(s)
static reserved
   initial reserved := ["break", "create", "do",
                        "else", "every", "if",
                        "next", "not", "then",
                        "to", "while"]
   if s == !reserved then fail
   return Ident(s)
end

procedure sb()                          # skip blanks
static blanks
   initial blanks := ' \t\n'
   tab(many(blanks))
   return
end

## gencode(parsetree) - generate ucode from the parse tree

record loop(type, brklab, nxtlab, ninstance)

procedure gencode(ptree)
local lab, lab1
static newlab, loopstk
   initial {
      loopstk := [ ]
      newlab := create "L"||(0 to 9)||(0 to 9)||(0 to 9)
      lab := @newlab
      write("mark ", lab)
      gencode(ptree)                    # generate code
      write("label ", lab)
      write("stop")
      write("label Flab")               # universal failure
      write("fail")
      write("label Clab")               # universal co-expression failure
      write("cofail")
      write("goto Clab")
      return
      }

   case type(ptree) of {
   "null" :
        write("push ")
   "integer" |
   "real"    |
   "string"  :
        write("push ",ptree)
```

```
"Binop" :
     {
     gencode(ptree.left)
     gencode(ptree.right)
     write("invoke ", ptree.opcode)
     }

"Unop" :
     {
     gencode(ptree.e)
     write("invoke ", ptree.opcode)
     }

"Noop" :
     write("invoke ", ptree.opcode)

"Break" :
     {
     if \loopstk[1] then {
        every 1 to loopstk[1].ninstance do
           write("unmark")
        if \ptree.e then {
           saveloop := pop(loopstk)
           gencode(ptree.e)
           push(loopstk, saveloop)
           write("goto ", loopstk[1].brklab)
           }
        else
           write("goto Flab")
        }
     else
        write("*** illegal context for break")
     }

"Compound" :
     {
     write("mark ", lab := @newlab)
     \(loopstk[1]).ninstance +:= 1
     gencode(ptree.left)
     \(loopstk[1]).ninstance -:= 1
     write("unmark")
     write("label ", lab)
     gencode(ptree.right)
     }
```

```
"Create" :
     {
     push(loopstk, &null)
     write("goto ", lab := @newlab)
     write("label ", lab1 := @newlab)
     write("pop")
     write("mark Clab")
     gencode(ptree.e)
     write("coreturn")
     write("fail")
     write("label ", lab)
     write("create ", lab1)
     pop(loopstk)
     }

"Every":
     {
     lab := @newlab
     push(loopstk, loop("every", lab, "Flab", 1))
     write("mark Flab")
     gencode(ptree.ctl)
     if \ptree.docls then {
        write("pop")
        write("mark Flab")
        loopstk[1].type := "everydo"
        loopstk[1].ninstance +:= 1
        gencode(ptree.docls)
        write("unmark")
        }
     write("fail")
     write("label ", lab)
     pop(loopstk)
     }

"Ident"  :
     write("pushv ", ptree.var)
```

```
"If"    :
        {
        if \ptree.ecls then
           write("mark ", lab := @newlab)
        else
           write("mark Flab")
        \(loopstk[1]).ninstance +:= 1
        gencode(ptree.ctl)
        \(loopstk[1]).ninstance -:= 1
        write("unmark")
        gencode(ptree.tcls)
        if \ptree.ecls then{
           write("goto ",lab1 := @newlab)
           write("label ", lab)
           gencode(ptree.ecls)
           write("label ",lab1)
           }
        }

"Next" :
        {
        if \loopstk[1] then {
           if loopstk[1].type ~== "every" then {
              every 1 to loopstk[1].ninstance-1 do
                 write("unmark")
              if loopstk[1].type ~== "everydo" then
                 write("unmark")
              }
           write("goto ", loopstk[1].nxtlab)
           }
        else
           write("*** illegal context for next")
        }

"Not"   :
        {
        write("mark ", lab := @newlab)
        \(loopstk[1]).ninstance +:= 1
        gencode(ptree.e)
        \(loopstk[1]).ninstance -:= 1
        write("unmark")
        write("fail")
        write("label ", lab)
        write("push 0")
        }
```

```
 "Or"   :
        {
        write("mark ",lab := @newlab)
        \(loopstk[1]).ninstance +:= 1
        gencode(ptree.left)
        \(loopstk[1]).ninstance -:= 1
        write("esave")
        write("pushr")
        write("goto ",lab1 := @newlab)
        write("label ", lab)
        gencode(ptree.right)
        write("label ",lab1)
        }
"Limit" :
        {
        gencode(ptree.right)
        write("limit")
        write("mark Flab")
        \(loopstk[1]).ninstance +:= 1
        gencode(ptree.left)
        \(loopstk[1]).ninstance -:= 1
        write("lsave")
        write("pop")
        write("pushr")
        }

 "Reval":
        {
        write("label ", lab := @newlab)
        write("mark Flab")
        \(loopstk[1]).ninstance +:= 1
        gencode(ptree.e)
        \(loopstk[1]).ninstance -:= 1
        write("chfail ", lab)
        write("esave")
        write("pushr")
        }
 "While":
      {
      lab := @newlab
      lab1 := @newlab
      push(loopstk, loop("while", lab1, lab, 1))
      write("label ", lab)
      write("mark Flab")
      gencode(ptree.ctl)
      write("unmark")
      if \ptree.docls then {
         write("mark ",lab)
         gencode(ptree.docls)
         write("unmark")
         }
      write("goto ", lab)
      write("label ", lab1)
      pop(loopstk)
      }

  default:
      write("*** unimplemented: ", image(ptree))
  }

  return
end
```

## Appendix B — A One-Stack Model Interpreter

The following program interprets ucode for a subset of Icon, using the one-stack model of goal-directed evaluation.

```
record coexpr(activator, spc, sstk)
record instance(passive, spc, estk)
record op(opcode, operand)
record var(varname)

global current, active, pc, result
global code, labels, vars

## main - interpret ucode

procedure main(args)
   initial current := coexpr(1, &null, list())
   assemble()
   eval()
end

## assemble ucode, resolving references

procedure assemble()
local cp
   labels := table()
   code := list()
   cp := 1
   every scan !&input using {
            tab(many(' '))
            metacode := tab(upto(' ')|0)
            tab(many(' '))
            metavalue := tab(0)
            } do {
      if metacode == "label" then
         labels[metavalue] := cp
      else {
         put(code, op(metacode, metavalue))
         cp +:= 1
         }
      }
   return
end
```

```
## eval - interpret ucode

procedure eval()
local metacode, metavalue
static opcodes
   initial {
      opcodes := table()
      opcodes["@"] := activ;          opcodes["write"] := out
      opcodes[":="] := assign;        opcodes["writes"] := outs
      opcodes["&"] := conjunc;        opcodes["read"] := in
      opcodes["+"] := add;            opcodes[">"] := more
      opcodes["-"] := subtract;       opcodes["<"] := less
      opcodes["*"] := multiply;       opcodes["="] := equal
      opcodes["/"] := divide;         opcodes[">="] := moreequal
      opcodes["%"] := remainder;      opcodes["<="] := lessequal
      opcodes["to"] := step
      }

   vars := table()
   pc := 1
   while 1 <= pc <= *code do {
      metacode := code[pc].opcode
      metavalue := code[pc].operand
      case metacode of {
         "chfail":           chfail(labels[metavalue])
         "cofail":           cofail()
         "coreturn":         coreturn()
         "create":           creat(labels[metavalue])
         "esave":            esave()
         "fail":             failure()
         "goto":             {pc := labels[metavalue]; next}
         "invoke":           opcodes[metavalue]()
         "limit":            limit()
         "lsave":            lsave()
         "mark":             mark(labels[metavalue])
         "pop":              pop(active.estk)
         "push":             push(active.estk, result := metavalue)
         "pushr":            push(active.estk, result)
         "pushv":            push(active.estk, var(metavalue))
         "stop":             return
         "unmark":           unmark()
         }
      pc +:= 1
      }
   return
end
```

```
# The following routines represent the primitive operations
#   of an Icon machine.

procedure creat(spc)
   push(active.estk, coexpr(&null, spc-1, list()))
   push(active.estk[1].sstk,
        instance(&null, &null, list()))
   return
end

procedure mark(flab)
   push(current.sstk, instance(active, flab, list()))
   active := current.sstk[1]
   return
end

# The routine failure() decrements the reactivation pc,
#  since the main interpreter loop in eval() is going
#  to increment it immediately.

procedure failure()
   pc := active.spc - 1
   pop(current.sstk)
   active := current.sstk[1]
   return
end

procedure unmark()
   popto(current.sstk, active.passive)
   active := current.sstk[1]
   return
end

procedure save()
   push(current.sstk,
        instance(active.passive, pc, copy(active.estk)))
   active := current.sstk[1]
   return
end

procedure esave()
   pop(current.sstk)
   push(current.sstk,
        instance(active.passive.passive, active.spc,
                 copy(active.passive.estk)))
   active := current.sstk[1]
   return
end
```

```
procedure limit()
   if deref(1) <= 0 then failure()
   return
end

procedure lsave()
   if (active.passive.estk[1] -:= 1) > 0 then
      esave()
   else
      unmark()
   return
end

procedure chfail(flab)
   current.sstk[one_above(active.passive)].spc := flab
   return
end

procedure activate(coexpr)
   current.spc := pc
   coexpr.activator := current
   current := coexpr
   pc := coexpr.spc
   active := current.sstk[1]
   return
end

procedure coreturn()
   current.spc := pc
   current := current.activator
   active := current.sstk[1]
   pc := current.spc
   push(active.estk, result)
   return
end

procedure cofail()
   current.spc := pc
   current := current.activator
   active := current.sstk[1]
   failure()
   return
end

procedure popto(stk, pointer)
   if /pointer then stop("bad pointer")
   while stk[1] ~=== pointer do pop(stk)
   return
end
```

```
procedure one_above(pointer)
local i
   i := 0
   while current.sstk[i+:=1] ~=== pointer
   return i - 1
end

## operators and functions

procedure conjunc()
   result := pop(active.estk)
   pop(active.estk)
   pusn(active.estk, result)
   return
end

procedure assign()
local variable, value
   value := dpop()
   variable := pop(active.estk)
   vars[variable.varname] := value
   push(active.estk, result := variable)
   return
end

procedure add()
   result := dpop() + dpop()
   push(active.estk, result)
   return
end

procedure subtract()
local left, right
   right := dpop()
   left := dpop()
   result := left - right
   push(active.estk, result)
   return
end

procedure multiply()
   result := dpop() * dpop()
   push(active.estk, result)
   return
end
```

```
procedure divide()
local left, right
   right := dpop()
   left := dpop()
   result := left / right
   push(active.estk, result)
   return
end

procedure remainder()
local left, right
   right := dpop()
   left := dpop()
   result := left % right
   push(active.estk, result)
   return
end

procedure step()
   if deref(2) <= deref(1) then {
      result := active.estk[2]
      active.estk[2] +:= 1
      if active.estk[2] <= active.estk[1] then
         save()
      pop(active.estk)
      pop(active.estk)
      push(active.estk, result)
      }
   else
      failure()
   return
end

procedure out()
   if type(active.estk[1]) == "var" then
       write(.vars[active.estk[1].varname])
   else
       write(active.estk[1])
   return
end

procedure outs()
   if type(active.estk[1]) == "var" then
       writes(.vars[active.estk[1].varname])
   else
       writes(active.estk[1])
   return
end
```

```
procedure in()
   push(active.estk, read())
   return
end

procedure more()
local left, right
   right := dpop()
   left := dpop()
   if result := (left > right) then
      push(active.estk, result)
   else
      failure()
   return
end

procedure less()
local left, right
   right := dpop()
   left := dpop()
   if result := (left < right) then
      push(active.estk, result)
   else
      failure()
   return
end

procedure equal()
local left, right
   right := dpop()
   left := dpop()
   if result := (left = right) then
      push(active.estk, result)
   else
      failure()
   return
end

procedure moreequal()
local left, right
   right := dpop()
   left := dpop()
   if result := (left >= right) then
      push(active.estk, result)
   else
      failure()
   return
end
```

```
procedure lessequal()
local left, right
   right := dpop()
   left := dpop()
   if result := (left <= right) then
      push(active.estk, result)
   else
      failure()
   return
end

procedure activ()
local left, right
   right := dpop()
   left := dpop()
   activate(right)
   push(active.estk, result := left)
   return
end

procedure deref(loc)
   if type(active.estk[loc]) == "var" then
      active.estk[loc] := vars[active.estk[loc].varname]
   return .active.estk[loc]
end

procedure dpop()
local tmp
   tmp := deref(1)
   pop(active.estk)
   return tmp
end
```

## Appendix C — A Modified One-Stack Model Interpreter

The following program interprets ucode for a subset of Icon, using the modified one-stack model of goal-directed evaluation.

```
record coexpr(pc, sp, active, activator, stk)
record op(opcode, operand)
record var(varname)

global current, result
global code, labels
global vars

procedure main()
   initial current := coexpr(1, -1, -1, &null, list(300))
   assemble()
   eval()
end

## assemble ucode, resolving references

procedure assemble()
local cp
   labels := table()
   code := list()
   cp := 1
   every scan !&input using {
            tab(many(' '))
            metacode := tab(upto(' ')|0)
            tab(many(' '))
            metavalue := tab(0)
            } do {
      if metacode == "label" then
         labels[metavalue] := cp
      else {
         put(code, op(metacode, metavalue))
         cp +:= 1
         }
      }
   return
end
```

```
## eval - interpret the ucode

procedure eval()
local metacode, metavalue
static opcodes
   initial {
      opcodes := table()
      opcodes["@"] := activ;          opcodes["write"] := out
      opcodes[":="] := assign;        opcodes["writes"] := outs
      opcodes["&"] := conjunc;        opcodes["read"] := in
      opcodes["+"] := add;            opcodes[">"] := more
      opcodes["-"] := subtract;       opcodes["<"] := less
      opcodes["*"] := multiply;       opcodes["="] := equal
      opcodes["/"] := divide;         opcodes[">="] := moreequal
      opcodes["%"] := remainder;      opcodes["<="] := lessequal
      opcodes["to"] := step
      }

   vars := table()
   while 1 <= current.pc <= *code do {
      metacode := code[current.pc].opcode
      metavalue := code[current.pc].operand
      case metacode of {
         "chfail":          chfail(labels[metavalue])
         "cofail":          cofail()
         "coreturn":        coreturn()
         "create":          creat(labels[metavalue]-1)
         "esave":           esave()
         "fail":            failure()
         "goto":            current.pc := labels[metavalue] & next
         "invoke":          opcodes[metavalue]()
         "limit":           limit()
         "lsave":           lsave()
         "mark":            mark(labels[metavalue])
         "pop":             spop()
         "push":            spush(result := metavalue)
         "pushr":           spush(result)
         "pushv":           spush(var(metavalue))
         "stop":            return
         "unmark":          unmark()
         }
      current.pc +:= 1
      }
   return
end
```

```
# The following routines represent the primitive operations
#   of an Icon machine.

procedure mark(flab)
local sact
   sact := current.sp
   spush(current.active)
   spush(&null)
   spush(sact)
   spush(flab)
   current.active := sact
   return
end

procedure save()
local sact
   sact := current.sp
   spush(current.stk[current.active])
   spush(current.active)
   spush(current.stk[current.active-2])
   spush(current.pc)
   every
      spush(current.stk[current.active-4 to sact+1 by -1])
   current.active := sact
   return
end

procedure esave()
local passive, inactive, ssp, spc
   passive := current.stk[current.active]
   inactive := current.stk[current.active-1]
   ssp := current.stk[current.active-2]
   spc := current.stk[current.active-3]
   current.sp := current.active
   spush(current.stk[passive])
   spush(\inactive|passive)
   spush(current.stk[passive-2])
   spush(spc)
   every spush(current.stk[passive-4 to ssp+1 by -1])
   return
end
```

```
# The routine failure() decrements the reactivation address,
#   since the main interpreter loop in eval() is going to
#   increment it immediately.

procedure failure()
   current.pc := current.stk[current.active-3] - 1
   current.sp := current.active
   if \current.stk[current.active-1] then                # reactivating
      current.active := current.stk[current.active-1]
   else                                                   # failing
      current.active := current.stk[current.active]
   return
end

procedure unmark()
   current.sp := current.stk[current.active-2]
   current.active := current.stk[current.active]
   return
end

procedure limit()
   if deref(1) <= 0 then failure()
   return
end

procedure lsave()
local top_passive
   top_passive := current.stk[current.active-2]
   if (current.stk[top_passive+1] -:= 1) > 0 then
      esave()
   else
      unmark()
   return
end

procedure chfail(flab)
local one_above
   one_above := current.stk[current.active-2]
   current.stk[one_above-3] := flab
   return
end

procedure creat(spc)
   spush(coexpr(spc, -1, -1, &null, list(100)))
   return
end
```

```
procedure activate(coexpr)
   coexpr.activator := current
   current := coexpr
   return
end

procedure coreturn()
   current := current.activator
   spush(result)
   return
end

procedure cofail()
   current := current.activator
   failure()
   return
end

procedure spush(value)
   current.stk[current.sp] := value
   current.sp -:= 1
   return value
end

procedure spop()
   current.sp +:= 1
   return .current.stk[current.sp]
end

## operators and functions

procedure conjunc()
   result := spop()
   spop()
   spush(result)
   return
end

procedure assign()
local variable, value
   value := dpop()
   variable := spop()
   vars[variable.varname] := value
   spush(result := variable)
   return
end
```

```
procedure add()
   result := dpop() + dpop()
   spush(result)
   return
end

procedure subtract()
local left, right
   right := dpop()
   left := dpop()
   result := left - right
   spush(result)
   return
end

procedure multiply()
   result := dpop() * dpop()
   spush(result)
   return
end

procedure divide()
local left, right
   right := dpop()
   left := dpop()
   result := left / right
   spush(result)
   return
end

procedure remainder()
local left, right
   right := dpop()
   left := dpop()
   result := left % right
   spush(result)
   return
end
```

```
procedure step()
local sp
   sp := current.sp
   if deref(2) <= deref(1) then {
      result := current.stk[sp+2]
      current.stk[sp+2] +:= 1
      if current.stk[sp+2] <= current.stk[sp+1] then
         save()
      spop()
      spop()
      spush(result)
      }
   else
      failure()
   return
end

procedure out()
local top
   top := current.stk[current.sp+1]
   if type(top) == "var" then
       write(vars[top.varname])
   else
       write(top)
   return
end

procedure outs()
local top
   top := current.stk[current.sp+1]
   if type(top) == "var" then
       writes(.vars[current.stk[top.varname]])
   else
       writes(top)
   return
end

procedure in()
   spush(read())
   return
end
```

```
procedure more()
local left, right
   right := dpop()
   left := dpop()
   if result := (left > right) then
      spush(result)
   else
      failure()
   return
end

procedure less()
local left, right
   right := dpop()
   left := dpop()
   if result := (left < right) then
      spush(result)
   else
      failure()
   return
end

procedure equal()
local left, right
   right := dpop()
   left := dpop()
   if result := (left = right) then
      spush(result)
   else
      failure()
   return
end

procedure moreequal()
local left, right
   right := dpop()
   left := dpop()
   if result := (left >= right) then
      spush(result)
   else
      failure()
   return
end
```

```
procedure lessequal()
local left, right
   right := dpop()
   left := dpop()
   if result := (left <= right) then
      spush(result)
   else
      failure()
   return
end

procedure activ()
local left, right
   right := dpop()
   left := dpop()
   activate(right)
   spush(result := left)
   return
end

procedure deref(loc)
   loc := current.sp+loc
   if type(current.stk[loc]) == "var" then
      current.stk[loc] := vars[current.stk[loc].varname]
   return .current.stk[loc]
end

procedure dpop()
local tmp
   tmp := deref(1)
   spop()
   return tmp
end
```

## References

Allision, Lloyd. "Phrase Structures, Non-Determinism, and Backtracking", *Information Processing Letters*, Vol. 7, No. 3 (April 1978) pp. 139-143.

Atkinson, Russ. *Toward More General Iteration Methods in CLU.* CLU Design Note 54, MIT, Project MAC (September 1975).

Bobrow, Daniel G. and Ben Wegbreit. "A Model and Stack Implementation of Multiple Environments", *Communications of the ACM*, Vol. 16, No. 10 (October 1973) pp. 591-603.

Budd, Timothy A. *An Implementation of Generators in C.* Technical Report TR 81-5, Department of Computer Science, The University of Arizona. (August 1981).

Conway, Melvin. "Design of a Separable Transition-Diagram Compiler", *Communications of the ACM*, Vol. 6, No. 7 (July 1963). pp. 396-408.

Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 4 (C Implementation for UNIX).* Technical Report TR 81-4, Department of Computer Science, The University of Arizona. (July 1981).

Coutant, Cary A. and Stephen B. Wampler. *A Tour Through the C Implementation of Icon; Version 4.* Technical Report TR 81-11, Department of Computer Science, The University of Arizona. (July 1981).

Dahl, Ole-Jahn and C. A. R. Hoare. "Coroutines", *Structured Programming*, Academic Press, London. (1972). pp. 184-193.

Dewar, Robert B. K. *SPITBOL Version 2.0.* Technical Report S4D23, Illinois Institute of Technology (February 1971).

Dewar, Robert B. K. and Anthony P. McCann. "MACRO SPITBOL - A SNOBOL4 Compiler", *Software—Practice and Experience*, Vol. 7, No. 1 (January 1977). pp. 95-113.

Dewar, Robert B. K., Arthur Grand, Ssu-Cheng Liu, Edmond Schonberg and Jacob T. Schwartz. "Programming by refinement, as exemplified by the SETL representation sublanguage", *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1 (July 1979). pp. 27-49.

Doyle, John Nicoll. *A Generalized Facility for the Analysis and Synthesis of Strings, and a Procedure-Based Model of an Implementation*, Master's Thesis, The University of Arizona, (1975).

Floyd, Robert. "Nondeterministic Algorithms", *Journal of the ACM*, Vol. 14, No. 4 (October 1967). pp. 636-644.

Gimpel, James F. *SITBOL: Version 3.0*, Technical Report S4D30b, Bell Telephone Laboratories, Inc., Murray Hill. (June 1973).

Griswold, Ralph E. "The SL5 Programming Language and Its Use for Goal-Directed Programming", *Proceedings of the Fifth Texas Conference on Computing Systems* (October 1976). pp 1-5.

Griswold, Ralph E. "The Use of Character Sets and Character Mappings", *The Computer Journal*, Vol. 23, No. 2 (May 1980). pp. 107-114.

Griswold, Ralph E. *Expression Evaluation in Icon*. Technical Report TR 80-21, Department of Computer Science, The University of Arizona. (August 1980).

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

Griswold, Ralph E., David R. Hanson and John T. Korb. "Generators in Icon", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 2 (April 1981). pp. 144-161.

Grune, Dick. "A View of Coroutines", *SIGPLAN Notices*, Vol 12, No. 7 (July 1977). pp. 75-81.

Hanson, David R. "Storage Management for an Implementation of SNOBOL4." *Software—Practice and Experience* Vol. 7, No. 2 (March 1977). pp. 179-192.

Hanson, David R. "Filters in SL5", *Computer Journal*, Vol. 21, No. 2 (May 1978). pp 134-143.

Hanson, David R. and Ralph. E. Griswold. "Language Facilities for Programmable Backtracking", *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, (August 1977). pp. 94-99.

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

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

Ichbiah, J. D. and S. P. Morse. "General Concepts of the Simula 67 Programming Language", *Annual Review in Automatic Programming*. Vol. 7, No. 1 (1972). pp. 65-93.

Klint, Paul. "An Overview of the Summer Programming Language", *Seventh Annual ACM Symposium on Principles of Programming Languages*, (January 1980). pp. 47-55.

Knuth, Donald E. "Computer Programming as an Art", *Communications of the ACM*, Vol. 17, No. 12 (December 1974). pp. 667-673.

Korb, John T. *The Design and Implementation of a Goal-Directed Programming Language.* Technical Report TR 79-11, Department of Computer Science, The University of Arizona. (June 1979).

Kriz, J., and Sandmayr, H. "Extension of Pascal by Coroutines and its Application to Quasi-parallel Programming and Simulation", *Software - Practice and Experience*, Vol. 10 (1980). pp. 773-789.

Lampson, B. W., Mitchell, J. G., and Satterthwaite, E. H. "On the Transfer of Control Between Contexts", *Programming Symposium*, B. Robinet, editor. Springer-Verlag, New York. 1974. pp. 181-203.

Lehmer, Derrick H. "Combinatorial Problems with Digital Computers", *Proceedings of the Fourth Canadian Mathematical Congress.* 1957. pp. 160-173.

Lemon, Michael J. *Coroutine PASCAL: Case Study in Separable Control.* Master's Thesis, Department of Computer Science, University of Pittsburgh. December 1976.

Lindstrom, Gary "Backtracking in Generalized Control Settings", *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1 (July 1979). pp. 8-26.

Lynning, E. in letter to the editor, *SIGPLAN Notices*, Vol. 13, No. 2 (February 1978), pp. 12-14.

McDermott, Drew V. and Gerald J. Sussman. "From PLANNER to CONNIVER - a Genetic Approach", *Proceedings of the FJCC.* 1972. pp. 1171-1179.

McIlroy, M. Douglas. *Coroutines.* Technical Report, Bell Telephone Laboratories. (May 1968).

Montanegro, Carlo, Giuliano Pacini, and Franco Turini. "A Model for Structured Parallel Processing in Block-Structured Programming Languages", *Programming Symposium*, B. Robinet, editor. Springer-Verlag, New York. 1974. pp. 350-361.

Morris, James H., Eric Schmidt, and Philip Wadler. "Experience with an Applicative String Processing Language", *Seventh Annual ACM Symposium on Principles of Programming Languages*, (January 1980). pp. 32-46.

Moses, J. "The Function of FUNCTION in LISP", *SIGSAM Bulletin*, No. 4, (July 1970). pp. 13-27.

Mylopoulos, John, Norman Badler, Lucio Melli, and Nicholas Roussopoulos. "1.PAK: A SNOBOL-Based Programming Language for Artificial Intelligence Applications", *Proceedings of the International Joint Conference on Artificial Intelligence*, No. 3, 1973. pp. 691-696.

Perlis, Alan J. "In Praise of APL: A Language for Lyrical Programming", *SIAM News*, (June 1977). pp. 44-47.

Prenner, Charles J., Jay M. Spitzen, and Ben Wegbreit. "An Implementation of Backtracking for Programming Languages", *SIGPLAN Notices*, Vol. 7, No. 11 (Nov 1972). pp. 36-44.

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

Shaw, Mary, Wm. A. Wulf, and Ralph L. London. "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", *Communications of the ACM*, Vol. 20, No. 8 (August 1977). pp. 553-564.

Smith, David C., and Horace J. Enea. "Backtracking in MLISP2", *Proceedings of the International Joint Conference on Artificial Intelligence*, No. 3, 1973. pp. 677-685.