The Implementation of Generators and Goal-Directed Evaluation in Icon*

Janalee O'Bagy

TR 88-31

August 11, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants CCR-8713690, DCR-8401831, and DCR-8502015.

This technical report has been submitted as a dissertation
to the faculty of the Department of Computer Science in
partial fulfillment of the requirements for the degree
of Doctor of Philosophy in the graduate college of
the University of Arizona.

TABLE OF CONTENTS

TABLE OF CONTENTS – Continued

ABSTRACT

Generators and goal-directed evaluation provide a rich programming paradigm when combined with traditional control structures in an imperative language. Icon is a language whose goal-directed evaluation is integrated with traditional control structures. This integration provides powerful mechanisms for formulating many complex programming operations in concise and natural ways. However, generators, goal-directed evaluation, and related control structures introduce implementation problems that do not exist for languages with only conventional expression evaluation. This dissertation presents an implementation model using recursion that serves as a basis for both an interpreter and a compiler.

Furthermore, in the case of the compiler, optimizations can be performed to improve the efficiency of Icon programs, mainly by reducing the general evaluation strategy whenever possible. The dissertation describes a compile-time semantic analysis used to gather information about the properties of expressions and how they are used at their lexical sites. The optimizations that can be performed using this information are illustrated in the context of the compiler model described in the dissertation.

CHAPTER 1

INTRODUCTION

Icon is a programming language designed for string and list processing, as well as for general high-level programming tasks [18]. Icon generalizes the traditional view of expression evaluation in which expressions evaluate to a single result. In Icon, an expression may produce a sequence of results [16]. Such expressions are called generators. Generators produce only one result at a time; alternatives are produced if the surrounding context demands them. Alternative computational paths are the basis for the expression evaluation mechanism of Icon, which is goal-directed. Goal-directed evaluation attempts to produce a result for each expression. Failure during expression evaluation causes alternatives to be taken.

The implicit nature of goal-directed evaluation combined with the generalized concepts of success, failure, and generators gives Icon its expressive power. The embedded evaluation mechanism and control structures allow concise expression of computation; fewer explicit control structures need be present in an expression. The challenge for the implementor is to design a coherent, well-integrated model for implementing the features of expression evaluation.

There have been several implementations of Icon based upon a definition of an Icon virtual machine, with an interpreter to execute the virtual machine instructions. However, in these interpreters the core of the expression evaluation mechanism is intricate and complicated and the simple properties of the underlying semantics are obscured.

This dissertation describes a new implementation model for the Icon virtual machine, based on recursion. This model for implementing goal-directed evaluation and generators is coherent and easy to understand and extend. Although efficiency is not a primary concern here, the recursive implementation does not sacrifice speed and it performs comparably to previous implementations.

The recursive model for implementing generators and goal-directed evaluation is not specific to the virtual machine. When applied to the Icon virtual machine, the recursive model results in an interpreter that gives a simple operational semantics for expression evaluation in Icon. The recursive model can also be used as the basis for compiling Icon expressions. The dissertation discusses how the recursive model is used to both interpret and compile Icon expression evaluation.

An evaluation regime, such as goal-directed evaluation, that is general and powerful may also be inefficient in its implementation as well, precisely because of the generality it imposes on the control behavior of programs. This is an important implementation issue for Icon, since it is possible to write programs that use expressions in a traditional way, expressions which do not take advantage of the full capabilities of goal-directed evaluation. Icon's strength as a programming language, in fact, is largely due to the way that it combines traditional control structures with goal-directed evaluation, and gives a programmer the advantages of both paradigms. Not all expressions, therefore, require the general mechanism for evaluation. Such expressions

should not pay for the cost of the general evaluation strategy.

Previous implementations of Icon have not addressed this issue; they employ the general strategy of evaluation at all times. This dissertation provides a method for detecting the unnecessary generality of expression evaluation. It describes the relationship between properties of expressions and their evaluation requirements, it gives a method for obtaining this information at translation time, and it illustrates its use in improving code generation for Icon programs. The optimizations made for expression evaluation are illustrated in the context of the compiler model described in this dissertation.

1.1 Generators and Goal-Directed Evaluation

In general, the term generator denotes a procedure or expression that can be repeatedly activated to produce a sequence of results. Like a coroutine, the internal state of a generator must be saved between the time the generator produces a result and the time it is resumed to continue its computation. Yet, like a procedure, the evaluation/return pattern of generators is hierarchical [3, 36].

Generators have been incorporated in several programming languages, for example IPL [34], SETL [37], and CLU [31]. Primarily, generators are used in these languages as an abstraction for processing elements of data structures. They are not, however, a fundamental aspect of the evaluation of expressions. For example, in CLU, a generator can be used only in the control portion of the language's for statement. More recently, experimental languages have been designed to incorporate generators in the style of Icon into C [6] and Pascal [12].

Goal-directed evaluation is a term used in connection with a variety of languages, including PLANNER [14, 25], and logic programming languages such as Prolog [29, 40]. Generally, goal-directed evaluation is a strategy that proceeds forward on an evaluation path in an attempt to satisfy a goal, and backtracks to a previous alternative path if the goal cannot be satisfied along the current path. Languages with goal-directed evaluation offer different linguistic mechanisms for expressing alternatives. Examples are patterns in SNOBOL4 and nondeterministic predicates in Prolog. Backtracking automatically examines the different control paths that are expressed implicitly in the program until the language's notion of success is achieved.

Goal-directed evaluation is a powerful control abstraction. It eliminates the need to specify an algorithm in detail, and in some languages, the declarative nature of the language combined with goal-directed evaluation subsumes any algorithmic specification of solutions altogether. On the other hand, a goal-directed process, though powerful, is also limiting—there is only one general strategy of control flow, and no way to pre-empt or refine it. Yet often, a search and backtrack algorithm is not an efficient method for finding a solution. It has been recognized that a goal-directed mechanism alone is not sufficient for general-purpose programming [15, 19, 42]. In logic programming languages, the cut primitive is often provided to inhibit backtracking, even though this primitive is not a logical construct [40].

Icon solves this problem by integrating goal-directed evaluation and traditional control structures in a way that is elegant and uniform. Generators provide the source of alternatives used by goal-directed evaluation. A programmer may use goal-directed evaluation when convenient, and yet also express algorithms directly by using the control structures of the language.

This kind of integration has been attempted more recently in languages designed to combine the logic and functional language paradigms [4]. The language Fresh is an example [38]. Because computation, and subsequently control flow, in Fresh is not based strictly on unification,

such an integration results in a failure-based expression language that closely resembles the evaluation of expressions in Icon, which is discussed further in the following section. Fresh concepts include generating functions, built by using alternation (which they call disjunction), bounding (which they call confinement), and a conditional control structure that bounds its control clause.

1.2 Expression Evaluation in Icon

As mentioned previously, an expression in Icon can produce a sequence of results. For example, find(s1,s2) is capable of producing all the positions at which s1 occurs as a substring of s2, and i to j is capable of producing the integers in order from i to j.

The *result sequence* for an expression consists of the results it is capable of producing. For example, the result sequence for 1 to 5 is {1, 2, 3, 4, 5}. An expression can have an empty result sequence. Examples are the comparison operation 1 = 2, and the function read() when at the end of the input file. An expression whose result sequence has at least one result *succeeds*. An expression with an empty result sequence *fails*.

Success and failure determine the behavior of control structures. For example, consider the if expression:

```
if i < max then write("out of bounds") else write("okay")
```

If the control clause succeeds, the then clause is evaluated; otherwise the else clause is evaluated. Similarly, a while loop is driven by success or failure of its control clause, as in:

```
while line := read() do
   write(line)
```

Evaluation based on success and failure allows concise formulation of computation. The while expression above, for example, can be written as:

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

An expression that is capable of producing more than one result is called a *generator*. A generator only produces more than one result if the context in which it is evaluated demands it. The implicit evaluation context for all expressions is goal-directed. Goal-directed evaluation attempts to produce a result for each expression. During evaluation, if an expression fails, the most recently suspended expression is resumed for a subsequent result. Consider, for example, the expression

```
i = find(s1,s2)
```

where the value of i is 6 and the result sequence for find is {4, 6, 12}. When the values 6 and 4 are compared, the expression fails. Goal-directed evaluation then resumes find, which produces its next result, 6. The comparison then succeeds.

In general, expressions are evaluated from left to right, and resumed in a last-in, first-out order during goal-directed evaluation. The result is cross-production evaluation. For example, if $expr_1$ has the result sequence {1, 2, 3}, and $expr_2$ has the result sequence {10, 20, 30}, the result sequence for

$$expr_1 + expr_2$$

is {11, 21, 31, 12, 22, 32, 13, 23, 33}.

An important semantic feature of Icon concerns the lexical scope of its goal-directed evaluation mechanism. Rather than having the entire program, or even a procedure body, consist of one monolithic expression in which backtracking is unlimited, Icon programs consist of separate *bounded expressions*. Backtracking is limited to the scope of a bounded expression. Once a bounded expression produces a result, it cannot be resumed to obtain another. To understand why bounding is necessary, suppose that the expression above is the control clause for an if expression:

if i = find(s1,s2) then $expr_1$ else $expr_2$

Since evaluation of i = find(s1,s2) succeeds (as described above), the then clause is selected. Now suppose that evaluation of $expr_1$ fails and that failure is allowed to propagate back into the control clause. At this point, find is resumed, but it has no alternative results that can make the comparison succeed. Hence the control clause fails and the else clause is chosen. Backtracking into the control clause is erroneous: both the then and else clauses could be evaluated, violating the natural semantics of if. For this reason, Icon bounds the control clause of the if.

Bounded expressions are the basic structural components of goal-directed evaluation in Icon. They begin a context for goal-directed evaluation of an expression and control the generation of results for expressions. Understanding where bounded expressions occur in the source language is important to a clear understanding of the implementation. Some examples of implicitly bounded expressions in control structures are listed below.

- The expression of a procedure body is bounded.
- All expressions in a compound expression {$expr_1$; $expr_2$; ...; $expr_n$ } are bounded, except $expr_n$.
- The control expression of an if expression is bounded.
- The control and do expressions of the traditional looping control structures are bounded.

Appendix A gives a complete list of all control structures of Icon and identifies their bounded expressions.

Since a procedure body expression is bounded, evaluation always takes place within a bounded expression. In the following expression, the bounded expressions are enclosed in rectangles to illustrate that bounded expressions become may be nested:

```
procedure p()
   while [expr1] do
      if [expr2] then [expr3]
end
```

Icon integrates goal-directed evaluation into conventional control structures with the discipline provided by bounding. In addition to the conventional control structures that are found in other languages, Icon has several control structures related to generators. These control structures are meaningless and unnecessary in a traditional imperative or functional language in which every expression produces exactly one result. One of these new control structures is alternation, $expr_1$ | $expr_2$, which generates the results of $expr_1$ followed by the results of $expr_2$. The result sequence for alternation is the concatenation of the result sequences of its arguments. For

example, the result sequence for

```
(1 to 5) | (8 to 10)
```

is { 1, 2, 3, 4, 5, 8, 9, 10}.

The repeated alternation control structure, |*expr*, generates the results of *expr* repeatedly. For example, the result sequence for

```
|(1 to 3)
```

is { 1, 2, 3, 1, 2, 3, ... }. While this result sequence is infinite, it is only a potential infinity that poses no computational problem. Generators produce only the number of results that their evaluation context requires in its attempt to succeed. An exception to the repeated generation of results occurs if the argument expression fails. In this case, the repeated alternation terminates. Only expressions whose result sequences depend on side effects can succeed at one time and fail at another. For example, the evaluation of read() produces a result for each line in the input file, but fails when there are no more lines. Consequently, the result sequence for |read() consists of all the lines of the input file.

A result sequence can be truncated by the *limitation* control structure, *expr* \ i, which limits *expr* to i results. For example, the result sequence for

```
|(1 to 3) \ 5
```

is { 1, 2, 3, 1, 2}.

The every control structure is used to iterate over all the results of an expression. For example,

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

writes every position at which s1 occurs in s2. The general form for the every control structure is:

every $expr_1$ do $expr_2$

For each result produced by $expr_1$, $expr_2$ is evaluated.

1.3 Implementation Issues

In an imperative or functional language where expressions produce a single result, interpreting expressions is straightforward. For example, expression evaluation can be accomplished by a depth-first traversal of a tree representation of the expression. There are many examples of simple, elegant recursive interpreters for languages such as Lisp and Scheme [1, 32]. Because expressions in these languages produce only one result, an interpreter can employ a simple recursive process of evaluating arguments, and performing the application of an operation to the arguments. Generators can be implemented by this simple process of recursive evaluation/application only if the implementation language itself has generators or some other general control abstraction such as coroutines or continuations. An interpreter for Icon written in Icon and making use of suspension (generation) is discussed in [17]. This approach is useful for considering semantics at a higher-level, but is impractical as an implementation technique, since it requires suspension in the implementation language.

This dissertation focuses instead on the implementation issues for Icon using a traditional procedural language such as C or Pascal. That is, the implementation language provides only

the the usual semantics for procedures and expressions.

Generators bring two issues to the implementation. First, generators must maintain local information between suspension and resumption. For example, in the expression

```
i = find(s1,s2)
```

find must keep track of its arguments and the current position in s2. In that sense, the primitive operations and functions of the language are not atomic, as they are in most languages. They are not routines or machine operations that perform a computation all at once. The information related to a suspended generator must be maintained in much the same way that procedure call information is maintained in traditional languages. That is, generators in Icon require state maintenance at the expression level that usually only occurs at the procedure level in other languages.

The second issue is subtler: Generators prolong the lifetime of temporary values. For example, in

```
i = find(s1,s2)
```

the operands of the comparison operation cannot be discarded when find produces its result. If find is resumed, the comparison is performed again with subsequent results from find(s1,s2), and the left operand must still be available. The solution to the prolonged lifetime of temporary values ultimately depends on the design of the virtual machine for Icon. The Icon virtual machine is a stack machine where operations take their operands from the stack and replace them by their results. Since operations consume their operands, they consume temporary values that may be needed later. Consequently, such operands must be replicated when generators are present. If the Icon virtual machine were a temporary register machine, temporary values would not be consumed by computations. Rather, the temporary register allocator would release a temporary only when its lifetime is complete. The lifetimes of temporaries are limited lexically by the implicit bounded expressions of control structures. A simple register allocator would release the temporary registers allocated for a bounded expression when it reaches the end of the bounded expression. A method to allocate temporary registers more efficiently within a bounded expression is a considerably more complicated problem that the allocation of temporaries for conventional expression evaluation, since it requires knowing when generators are present, and the extent of the generators.

In addition to these issues related to a generator's computation state, a generator implicitly provides a control backtracking point for goal-directed evaluation. Within a bounded expression, control backtracking points introduced by generators are accumulated as the expression is evaluated. The backtracking points obey stack protocol and the last generator evaluated is the first one to be resumed. Generative control structures also introduce control backtracking points and may also require additional state information. The limitation control structure, for example, requires a method for keeping track of the number of results an expression produces.

Finally, bounding restricts the use of backtracking points that accumulate during evaluation. Because bounded expressions are nested, their respective demarcations—the markings that tell how far back to pop the stack—are nested during evaluation as well.

These implementation issues can be summarized as follows:

- The backtracking points due to generators must be maintained.
- The bounding information due to control structures must be maintained.
- The state information for generators must be preserved.
- The lifetime of temporary values must be extended in the presence of generators.

For the purposes of dealing with expression evaluation in Icon, there are three properties of expressions that are important. Expressions that always produce a single result are equivalent to expressions found in conventional languages. Such expressions are called *monogenic*. Expressions that can fail to produce a result are fundamental to driving goal-directed evaluation. Such expressions are called *conditional*. Finally, expressions that can produce more than one result introduce state retention and are called *generators*. Appendix B lists the operations and functions in Icon and their corresponding properties.

1.4 Previous Implementation Models

As mentioned previously, several implementations of Icon have been developed since the original version written by the designers of Icon. This section briefly discusses three different versions that use similar Icon virtual machines as the basis of implementation. They differ significantly in their approaches to solving the following problems: maintaining the failure points of bounded expressions and generators, detecting failure and resuming generators, maintaining local state information for generators, and maintaining temporary values. The various approaches result in different management of the system stack (the stack of the implementation language) and other auxiliary stacks. The virtual machine itself is discussed in Chapter 2, and is not needed to understand the following discussion.

Versions 1 and 2. The original implementation of Icon, written in Fortran [16, 28], uses two stacks. The interpreter stack holds active procedure frames and the temporary Icon values resulting from expression evaluation. A second stack, called the control stack, holds control and state information for goal-directed evaluation.

During evaluation, a global variable holds the failure label for the current bounded expression. The failure label indicates where evaluation should continue when failure occurs. When a bounded expression is evaluated, the current failure label is saved on the control stack and the new failure label is assigned to the failure variable. The heights of the interpreter and control stacks are also saved on the control stack. In the absence of generators, if failure occurs during evaluation, the current failure label is used to continue evaluation. Icon values on the interpreter stack due to the current bounded expression are removed by restoring the interpreter stack height to its original value, and the most recent failure label is restored from the stack.

Generators use the control stack to maintain local state between suspension and resumption. Generators are written so that each *call* to the generator produces the next result. Thus, they follow a programming convention that determines whether a call is the initial call, or a call due to resumption. After a generator produces a result, it saves its local state on the control stack and also copies the temporary Icon values from the interpreter stack to the control stack. By using the most recently stacked height, only the values relevant to the current expression are saved. The generator then pushes a label on the stack that points to the place in the code where the generator is called.

When failure occurs, the failure routine checks for a generator. If there is a generator, the failure routine resumes the most recently suspended generator by restoring the information from the control stack. Once the local state of the generator and the temporary Icon values are restored

on the interpreter stack as they were at the time the generator suspended, control is transferred to the label stored by the generator, and the generator is called to produce its next result.

This implementation requires that temporary Icon values be copied twice, once to be saved on the control stack, and again when the values are restored to the interpreter stack. Furthermore, the code of a generator is highly specialized and differs from other routines in the run-time system.

Versions 3 through 5. Versions 3 through 5 of Icon [20,45,46], which are written in C and assembly language, combine the interpreter and control stacks on the system stack used by C. All information for active procedure frames, bounded expressions, generators, and temporary Icon values is maintained on the same stack. This requires assembly language code to augment the C code, since the system stack is manipulated in nonstandard ways throughout interpretation.

Information for expression evaluation is maintained in two types of frames: expression frames and generator frames. These frames are stored on the system stack and global pointers are maintained to point to the current expression and generator frames during execution. Whenever a bounded expression is entered, a new expression frame is created. Expression frames hold the values of the previous expression and generator frame pointers and a failure continuation associated with the current bounded expression. On the other hand, when a generator suspends, a generator frame is created. Generator frames hold the previous frame pointer values and a failure continuation for the generator. Following the generator frame, the Icon temporary values relevant to the current expression are copied on the stack. The local state of the generator is maintained by keeping its activation frame on the stack and branching back to the main interpreter routine.

When failure occurs, the interpreter examines the current values of the expression and generator frame pointers. If a generator is present, the interpreter restores information saved in the generator frame and transfers back to the suspended generator. If there is no generator, the interpreter removes the current expression frame and continues execution at the failure continuation stored in the expression frame.

The interaction between generator and expression frames in the implementation is rather subtle and confusing. For example, some control structures, such as alternation and limitation, cause both expression and generator frames to be created. The conceptual basis of expression evaluation is obscured by the interleaving of the frames. Furthermore, the interpreter and all routines associated with expression evaluation are written in assembly language. This implementation is by far the most complex in its treatment of expression evaluation.

Version 6. Version 6 of Icon [22] is similar to Version 5, and is also written in C, but does not use assembly language to implement expression evaluation. This transformation was accomplished by using recursion in the implementation of generators. Version 6 also makes use of an interpreter stack for expression evaluation information in addition to the system stack used by C.

Whereas Versions 3 through 5 retained the local state of a generator by leaving its activation record on the system stack and branching to the interpreter (using assembly language), in Version 6 the generator calls the interpreter recursively, avoiding assembly language. Recursion is used only for generative operators and functions, however, and not for generative control structures. The values of Icon expressions are maintained on the interpreter stack and operations use the values on the interperter stack for their arguments. Version 6 uses expression and generator frames to implement the control flow of goal-directed evaluation. The frames have the same structure and meanings as in Version 5, but are maintained on the interpreter stack, interleaved

with the Icon values, instead of on the system stack.

1.5 Comments

The fundamental problem with these implementations is that the conceptual basis for expression evaluation is complicated. The intricacy arises because the implementations distinguish between the failure control points for bounded expressions and those for generators. Thus separate mechanisms, for example, the frames of Versions 3 through 6, are required for each. When failure occurs, the actions taken differ depending on whether or not generators are present. Furthermore, the control information that is explicitly constructed for bounded expressions and generators must be explicitly removed.

The recursive model for implementing expression evaluation simplifies goal-directed control flow by treating failure control points due to bounded expressions and generators in a uniform manner. By synthesizing the failure points, all control information for goal-directed evaluation can be kept implicitly by recursion. The interpreter presented in Chapter 2 introduces a model of implementing generators and goal-directed evaluation that is separate from the virtual machine. It provides a conceptual basis for maintaining control information for Icon's expression evaluation that can be applied to interpretation of virtual machine code, or used directly to compile code for Icon. Using the recursive method as the basis for compiled code is discussed in Chapter 3.

Chapter 4 discusses optimization of expression evaluation. It focuses on the properties of bounding and generating. It shows how to gather information during translation, and how to use this information during code generation to eliminate unnecessary generality and to improve the generated code based on the properties of the expression at hand. Examples of programs that are improved by these methods are given.

Chapter 5 discusses the performance of the recursive interpreter as compared to Version 6 of Icon. It continues with an evaluation of the compiled code with optimizations. Chapter 5 concludes with suggestions for further work and summarizes the work in this dissertation.

CHAPTER 2

AN INTERPRETER FOR ICON

High-level programming languages are often implemented with interpreters. An interpreter is a convenient abstraction, allowing implementation at a conceptual level that is closer to the language than the instructions of an actual machine. One approach to implementing Icon expression evaluation is to translate source into a simple virtual machine language and then interpret the virtual machine. There are many possible designs for a virtual machine for Icon, and further, given a virtual machine instruction set, many possible ways to implement the virtual machine.

Several implementations of Icon are based on a virtual machine originally designed by Hanson and Korb [28]. The virtual machine is stack-based and has instructions that relate directly to source-level constructs. The stack nature of the machine avoids the issue of register allocation for temporary values, since temporary values are not assigned to specific locations but rather accumulate on a stack. The implementations of the virtual machine have varied significantly over the years. The disadvantage of previous implementations is that they are complicated and *ad hoc* in their treatment of expression evaluation. This chapter presents a new interpreter for the virtual machine that is clear and uniform.

2.1 The Virtual Machine

In the stack-based virtual machine, expressions are translated to postfix notation. Operations get their arguments from a stack and replace the arguments with the result of the operation. All types of operations—monogenic, conditional, and generative—are translated in the same way. For example, the code for i + 5 is:

```
var   i
int   5
plus
```

Likewise the code for 1 to 10 is:

```
int   1
int   10
to
```

The handling of failure and generators is left to the interpreter for the virtual machine code.

The basic unit of control for goal-directed evaluation is the bounded expression. Evaluation always takes place within a bounded expression. In the virtual machine, a bounded expression $expr_i$ is represented by the following code:

```
        mark L1
        code for expr1
        unmark
          :
          :
L1:
```

If $expr_1$ fails, execution continues at the code at label L1, the failure continuation of the bounded expression. If $expr_1$ produces a result, the unmark instruction is reached and execution continues at the instruction following the unmark. Conventional control structures are translated into code consisting of explicitly bounded expressions with the appropriate failure labels. For example, the code for the compound expression { $expr_1$; $expr_2$; $expr_3$ } is

```
        mark L1
        code for expr1
        unmark
L1:
        mark L2
        code for expr2
        unmark
L2:
        code for expr3
```

Similarly, the expression if $expr_1$ then $expr_2$ else $expr_3$ is translated into

```
        mark L1
        code for expr1
        unmark
        code for expr2
        goto L2
L1:
        code for expr3
L2:
```

Note that only the control clause is bounded in the if expression. The selected expression is evaluated within whatever context the if expression occurs.

A looping control structure is used to repeatedly evaluate an expression as long as the control clause succeeds. The loop expression fails when its control clause fails. A different form of the mark instruction, which does not have an explicit failure label, is used for the control clauses of loops. This instruction is mark0, which transmits failure to the surrounding context. For example, while $expr_1$ do $expr_2$ consists of two bounded expressions and is translated into:

```
L1:
        mark0
        code for expr1
        unmark
        mark L1
        code for expr2
        unmark
        goto L1
```

The repeat and until expressions are similar.

The control structures alternation, repeated alternation, and limitation do not bound their arguments and therefore do not use the mark instruction. Each control structure is translated into instructions specialized for that structure. For example, the code for alternation is:

```
        alt   L1
        code for expr1
        goto  L2
L1:
        code for expr2
L2:
```

The virtual machine representations of the remaining control structures are presented in later sections.

There are two important aspects of the virtual machine to consider. First, notice that control flow is based on where to continue on failure. All traditional control structures begin with a mark *label* instruction that specifies where to continue if failure propagates back to the mark instruction. Likewise, generative control structures and operators provide failure continuations. The alternation control structure specifies a failure continuation explicitly in its virtual machine representation. A generative operator such as to has an implied failure continuation. If to is resumed for a subsequent result, it computes its next result and begins evaluation at the instruction immediately following the to instruction. Consequently, an expression that can fail, such as the comparison operation <, is not given a failure continuation explicitly in the virtual machine representation. The most recent control structure or generator provides the failure continuation dynamically for a conditional expression.

Second, because the virtual machine is a stack machine, the temporary values on the stack are consumed when operations are successfully performed. For example, an addition operation consumes the top two values on the stack. Since generators may cause temporary values to be reused, the interpreter must have a method to save the temporary values.

2.2 The Recursive Model

The recursive interpreter focuses on the notion of a place to continue on failure. Failure continuations are introduced at control points. The control points, or control decisions, are introduced by control structures and generators. (Note that some generators are control structures and others are operations or functions.) A generator and a bounded expression introduced by a traditional control structure such as while are semantically very different, yet each provides a place to continue when failure occurs. The generator's failure continuation provides an alternative; the bounded expression's failure continuation provides the next sequential execution point of the program. In the recursive model, they are handled similarly.

Recursing when generators are encountered implicitly preserves both the internal state of the generator and control information for last-in, first-out resumption. Similarly, recursion for bounding indicates that most recent failure continuation is not a resumption point but a structured control flow point. In either case, recursion maintains all control information uniformly, simplifying the evaluation process. The recursive interpreter is mainly described by the interplay between recursing at failure continuation points and returning, either to resume generators or to discard them.

Conceptually, an expression is evaluated in a goal-directed *evaluation context*. The main component of the evaluation context consists of the failure continuation. Bounded expressions,

generators, and generative control structures change the current evaluation context, since each of these constructs provides a new failure continuation point. Whenever the interpreter encounters an expression that provides a new failure continuation, it saves the failure continuation and calls itself to provide a new context for evaluation. If the expression is a generator, the interpreter also replicates the appropriate values on the stack, since these values may be used again if generators are resumed. If failure occurs in a subsequent context, the interpreter returns to the previous context with a signal to resume generators. Execution then continues at the failure continuation of that context. On the other hand, if the end of a bounded expression is reached, the interpreter returns a signal indicating that the bounded expression is to be removed. This unwinds all levels of recursion built up during the evaluation of the bounded expression and achieves the effect of limiting the expression to one result during goal-directed evaluation.

The expression evaluation context is represented primarily in the interpreter by two local variables, one for the failure continuation point and one for a pointer into the stack holding temporary values that identifies the beginning of the values relevant to the expression. To implement generators and goal-directed evaluation, a conventional interpreter for expressions evaluated on a stack is augmented with these two state variables to represent that evaluation context. The interpretive process follows the method outlined above.

2.3 The Interpreter

An interpreter for Icon based on the ideas discussed above can be implemented in any traditional imperative language. The model places few demands on the implementation language, since expression evaluation is modeled as a simple call/return control pattern. The interpreter for Icon is implemented in C. Here, the interpreter is presented in simplified form in order to keep the discussion at a higher conceptual level than detailed C code would allow. To further simplify the explanation, expression evaluation is discussed without consideration of Icon procedure invocation and local variables. These topics are relatively uninteresting and are implemented in the standard way by allocating space for Icon local variables on the stack and maintaining a pointer to the local environment. Procedures and local variables are discussed briefly at the end of the chapter. In all the examples of Icon expressions that follow, variables are assumed to be global variables.

The stack used by the virtual machine is called the *expression stack*; it holds the temporary values created during expression evaluation. Representing Icon values is an interesting issue in itself, since variables are not typed in Icon and a variable can be assigned any value during its lifetime. This problem is handled by representing all values uniformly by a *descriptor*. A descriptor consists of two parts, one to specify the type, the other to specify the value. For simple atomic values such as integers, the value portion of the descriptor represents the data for the value directly. For structured values and strings, the value of the descriptor is a pointer to data representing the value. For the purposes of describing expression evaluation in this dissertation, a descriptor is defined in C as follows:

```
struct descrip {
   int type;
   union {
      int integer;
      char *sptr;
      } value;
   };
```

The actual definition of a descriptor, which is also a two-word entity, is given in [22].

The interpreter uses the following global variables:

- icode—an array holding the virtual machine instructions
- ipc—an index into icode
- stack—the expression (virtual machine) stack used to hold Icon values
- iglobals—an array consisting of the global variables of the Icon program

The variable icode is an array of integers, each of which denotes a virtual machine instruction or an operand of an instruction. This array is filled at interpreter initialization time. The ipc is an index into the icode array. Both stack and iglobals are arrays of descriptors.

Each invocation of the interpreter is an evaluation context for an expression. Variables local to the interpreter are used to maintain the expression evaluation context. The variable ep points to the portion of the expression stack where the values relevant to the current expression begin, and sp points to the current top of the expression stack. The variable fipc is the failure continuation for the current expression; it is simply an index into icode.

Invocations of the interpreter accumulate as new expression contexts are encountered. The interpreter returns in two situations: when failure occurs or when control reaches the end of a bounded expression. The interpreter returns a value that informs the invoking context how to respond to the outcome. The signal Resume is returned when an expression fails and indicates that goal-directed evaluation must resume suspended generators. The signal Clear is returned when the end of a bounded expression is reached, which indicates that goal-directed evaluation in the most recent bounded expression has succeeded and is complete.

The structure of the interpreter is:

```
int icode[Codesize];
int ipc;
struct descrip iglobals[Globals];
struct descrip stack[StackSize];

interp(ep, sp)
int ep, sp;
{
   int fipc, signal, newsp;
```

```
    for ( ; ; )
        switch (FetchInst()) {
            case Var:
                :
                :
            case Int:
                :
                :
            case Plus:
                :
                :
            case Mark:
                :
                :
            }
}
```

The interpreter embodies the fetch-decode-execute cycle for the Icon virtual machine. The FetchInst macro increments the ipc and returns the instruction pointed to by its previous value. A corresponding FetchOpnd macro returns the operand of an instruction.

Simple instructions, such as those for literals, push values on the expression stack. For example, the code in the interpreter for the virtual machine instruction int is:

```
case Int:
    stack[++sp].type = INT;
    stack[sp].value.integer = FetchOpnd();
    break;
```

When an Icon variable is referenced, the interpreter pushes a descriptor for that variable on the stack. The value portion of the descriptor is the offset of the variable in the iglobals array. The code for var is:

```
case Var:
    stack[++sp].type = VAR;
    stack[sp].value.integer = FetchOpnd();
    break;
```

Variables are dereferenced when required by context. For example, operations dereference their operands. The DeRef macro dereferences a variable by looking up the corresponding value in iglobals and replacing the variable on the stack with its value.

An operation computes its result using values on the top of the expression stack as operands. Other details of the code depend on whether the operation is monogenic, conditional, or generative. A monogenic operation does not effect control flow. The code for the plus instruction typifies a monogenic operation:

```
case Plus:
    DeRef(sp-1);
    DeRef(sp);
    stack[sp-1].value.integer =
        stack[sp-1].value.integer + stack[sp].value.integer;
    sp--;
    break;
```

Notice that the arguments are replaced by the result of the addition and that sp is decremented.

Execution continues at the beginning of the interpreter loop.

When control decisions are encountered, the current context of the expression is saved by a recursive call to the interpreter; execution then continues in the new context. Since expression evaluation always takes place within a bounded expression, the details of control flow are explained beginning with bounded expressions.

2.3.1 Bounded Expressions

An expression is bounded in order to control the generation of its results. When a bounded expression produces a result, its computation is complete and any information related to it is removed. This information is of two kinds: the values on the expression stack that accumulate during evaluation of the bounded expression, and the recursive invocations of the interpreter due to generators within the bounded expression.

The ep points to the base of the expression stack for the expression currently being evaluated. A bounded expression does not need to be connected with the values that may currently reside on the expression stack; when evaluation begins for a bounded expression, ep is adjusted to point to a "fresh" portion of the stack.

As shown previously, a bounded expression *expr*$_i$ occurs as code surrounded by the instructions mark and unmark:

```
        mark L1
        code for expri
        unmark
           :
           :
L1:
```

The mark instruction "marks" the boundary for control backtracking and the label L1 is the failure continuation for *expr*$_i$. If the expression succeeds, the unmark instruction is reached and the interpreter removes the context of *expr*$_i$. The code for mark follows:

```
case Mark:
   fipc = FetchOpnd();
   if (interp(sp+1,sp) == Resume)
      ipc = fipc;
   break;
```

At the mark instruction, the failure continuation L1—an index into icode—is saved in fipc. The interpreter is called as interp(sp+1,sp), making the ep of the new context point to the first unused portion of the expression stack. The instructions of the bounded expression are interpreted in the new context.

The code for unmark is simply

```
case Unmark:
   return Clear;
```

If evaluation reaches the unmark instruction, the expression has succeeded and its context is to be removed. Notice that since ep and sp are local variables in the interpreter, when unmark causes a return, the expression stack is effectively popped as well as the C calling stack. Execution continues at the instruction following the unmark instruction.

2.3.2 Failure

Failure is communicated by returning a signal that indicates failure. In general, left-to-right evaluation of expressions causes recursion at control decision points, and failure causes the interpreter to return to the most recent control decision point. If there are no generators, the most recent control point is the failure continuation of the current bounded expression.

A conditional operation introduces the possibility of failure during evaluation. The interpreter code for numerical comparison operation < illustrates how failure can occur during evaluation

```
case NumIt:
   DeRef(sp-1);
   DeRef(sp);
   if (stack[sp-1].value.integer >= stack[sp].value.integer)
      return Resume;
   stack[sp-1].value.integer := stack[sp].value.integer;
   sp--;
   break;
```

Notice that if the operation succeeds, the evaluation actions are similar to those implementing a monogenic operation. However, if the operation fails, interp returns Resume to continue goal-directed evaluation in a previous context.

2.3.3 An Example

To illustrate this type of failure-based evaluation, consider evaluating the if expression whose result is the larger of two integers:

```
if i < j then j else i
```

The corresponding virtual machine code is:

```
        mark L1
        var  i
        var  j
        numlt
        unmark
        var  j
        goto L2
L1:
        var  i
L2:
```

At the mark instruction, the interpreter saves the failure continuation L1 and invokes itself with new ep and sp values to establish a new context. In the new context, the variables i and j are pushed onto the stack and the comparison operation is performed. For the moment, suppose that i *is* less than j. Then the comparison succeeds and numlt decrements the sp, leaving j on the top of the stack as the result.

The interpreter then executes the unmark instruction. This indicates the end of a bounded expression, whose context is to be removed. The context consists of the values on the stack from the ep and invocations of the interpreter caused by evaluating the bounded expression. In this case, the evaluation of i < j leaves only one result on the stack and does not incur any new invocations of the interpreter.

After the return, control returns to the interpreter instance at the Mark case. Since the signal is Clear, the failure continuation for the bounded expression is not used. Execution continues at the current value of the ipc, which points to the instruction following the unmark. As execution continues, j is pushed on the stack and it becomes the result of the if expression.

Now consider the same example where i is greater than or equal to j so that i < j fails. Execution proceeds as before up to the numlt instruction. Since the comparison fails, numlt returns the signal Resume. Control returns to the code in the interpreter at the Mark case. Since the signal is Resume, the failure continuation for the bounded expression is used to continue execution. The fipc points to the code at L1 and execution continues at the icode instruction var i, making this the outcome of the if expression.

In the absence of generators, failure is simple and merely causes the context of the bounded expression to be removed and execution to continue at the failure continuation associated with the bounded expression. The expression fails, but goal-directed evaluation has no alternatives, since the expression does not have generators.

The next section discusses generators, which introduce alternatives during goal-directed evaluation. Resumption of generators is straightforward and follows naturally from the method of recursing at failure continuation points.

2.3.4 Generators

A generator provides an alternative computational path during goal-directed evaluation. Put another way, its failure continuation provides goal-directed evaluation with the possibility to compute a different result that might cause a computation to succeed where the previous computation failed. Recursion is the basic mechanism used to encode failure continuations for control backtracking. The interpreter makes no distinction between the failure continuations of bounded expressions and those of generators; it maintains them similarly.

However, besides recursing to stack its failure continuation, a generator must also address the lifetime problem of temporary Icon values (see Chapter 1). In a given context of evaluation, the ep points to the base of the values on the expression stack that are relevant to the current expression context. Therefore, just before a generator invokes the interpreter, it copies the values on the expression stack from the current ep to the sp. In the new context, ep then points to the base of the replicated values. Evaluation in the new context uses only the values from its ep; the previous values on the expression stack are left intact. Replicating the values extends the lifetime of the temporary Icon values and makes them available again if the generator is resumed.

All varieties of generators in Icon—operators, functions, generative control structures, and Icon procedures—are implemented in the same way. Every form of suspension establishes a new failure continuation, replicates values on the expression stack, and calls the interpreter recursively to stack its failure continuation and establish a new context.

As in the code for mark, a generator checks the signal returned by the interpreter and resumes only if appropriate. If the generator is not to be resumed, the signal is propagated by returning it to a previous context of the interpreter, thus removing the context of the generator.

The alternation control structure is a simple generator that illustrates the basic actions of a generator. The virtual machine code for the expression $expr_1 \mid expr_2$ is

```
        alt   L1
        code for expr1
        goto  L2
L1:
        code for expr2
L2:
```

The label L1 is the failure continuation of the alternation expression. If goal-directed evaluation resumes the alternation expression, evaluation continues with the code for the alternative expression $expr_2$. The code for alternation is:

```
case Alt:
   fipc = FetchOpnd();
   newsp = copy(ep,sp);
   signal = interp(sp+1,newsp);
   if (signal == Resume)
      ipc = fipc;
   else
      return signal;
   break;
```

The function copy replicates the values on the expression stack and returns the new value of the stack pointer after the copy. Alternation saves the failure continuation, copies the top portion of the stack, and invokes the interpreter with values of ep and sp that point to the replicated portion. When the interpreter returns, alternation uses its failure continuation only if another result is required.

Consider evaluating the expression i < (j | k), which succeeds if either j or k is greater than i. The code for the expression is:

```
        var   i
        alt   L2
        var   j
        goto  L3
L2:
        var   k
L3:
        numIt
```

After the first var instruction is evaluated, the alternation instruction is evaluated. Alternation fetches and saves the failure continuation and replicates the values of the current context. This may include many values besides the variable i, depending on the lexical context of the expression i < (j | k) in the source code. Execution continues in a new context at the var j instruction. The goto avoids evaluating the alternative expression.

At this point, the numIt is executed. There are two important points to notice. First, numIt operates on replicated values. Any computations using values on the expression stack in this evaluation context do not affect the contents of the stack as they were when alternation was first encountered. Secondly, the most recent failure continuation is due to the alternation control structure; if the comparison fails, control returns to the code in the interpreter for the Alt case and its failure continuation is used to continue evaluation at the alternative expression.

The uniformity of the interpretive process makes goal-directed evaluation straightforward. In the example above, if the first attempt of the comparison succeeds, the result of the expression i < (j | k) becomes the value of j. The context of the alternation expression in the interpreter remains until the end of the bounded expression in which it occurs is reached. At that point, the alternation context is removed by a Clear signal. On the other hand, if the comparison fails or if failure occurs in a subsequent computation, a Resume signal is returned to alternation and execution continues at its failure continuation. The comparison is then made with the second alternative.

2.3.5 Generative Operators and Functions

The distinction between operators and functions is syntactic only, as far as expression evaluation is concerned. This section describes the implementation of the generator *expr*$_1$ to *expr*$_2$. Generative functions such as find are implemented in the same way.

The code generated for *expr*$_1$ to *expr*$_2$ is

```
code for expr1
code for expr2
to
```

The general actions of the to generator mimic those of alternation. There are only two new observations to be made about a generative operator. First, a failure continuation is not given explicitly as an argument to the to instruction. The failure continuation for to is simply whatever instruction follows it. Second, the arguments for to are on the top of the expression stack during evaluation of to. There is no need to replicate the *arguments* of to if it suspends. Therefore, unlike alternation, which copies from ep to the current sp, to copies from the current ep to the value just preceding its first argument. After the replicated values, to pushes the result of its computation on the stack. In that way, the previous values on the expression stack are properly connected with the result of the to operation.

The code for to follows:

```
case To:
   fipc = ipc;
   DeRef(sp-1);
   DeRef(sp);
   from = stack[sp-1].value.integer;
   limit = stack[sp].value.integer
   while (from <= limit) {
      newsp = copy(ep,sp-2);
      stack[++newsp] = INT;
      stack[newsp].value.integer = from;
      signal = interp(sp+1,newsp);
      if (signal != Resume)
         return signal;
      ipc = fipc;
      ++from;
      }
   return Resume;
```

2.3.6 Repeated Alternation

Repeated alternation requires an extension of the techniques seen so far. Briefly, the problem is that this control structure requires knowing information that is hidden in the levels of recursion; communicating by signals is insufficient. Fortunately, the method used to implement this control structure is simple and does not burden the interpreter with additional mechanisms and state information. Another control structure, limitation, is similar and requires the same technique.

In evaluating |*expr*$_1$, if repeated alternation did not check for failure of *expr*$_1$, the program could loop. Hence, repeated alternation must be aware of the outcome of evaluating *expr*$_1$. The generated code for the repeated alternation expression |*expr*$_1$ is:

```
repalt
code for expr1
contrep
```

Notice that contrep is executed only if *expr*$_1$ succeeds; if it fails, contrep is not reached.

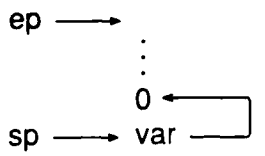
The difficulty in implementing repeated alternation is that the two instructions repalt and contrep must communicate. So far, the method used for communicating has been returning signals between contexts of interpretation. However, contrep cannot return to the context at repalt, since returning would remove contexts due to suspended generators in *expr*$_1$. For example, if the repeated alternation expression is 1 to 10, and if contrep returns, the invocation of the interpreter due to the generator to would be unwound and the generator would be removed prematurely.

A possible solution is to use a global variable. If contrep sets a global variable that indicates the expression succeeded, then when repalt regains control it could look at the value of the global variable to know if the expression succeeded. However, repeated alternation expressions may be nested. A global state variable must be maintained across expression contexts, being saved and restored at each invocation and return. Furthermore, the limitation control structure also requires similar communication. Thus, a second "state" variable would have to be introduced for it as well.

To avoid burdening the implementation with extra state variables, the interpreter uses a simpler, but computationally equivalent method. It makes the next available stack position the value of a variable used for communication for an instance of repeated alternation. In other words, each instance of repeated alternation dynamically allocates its own local variable on the expression stack for communication. The code below illustrates the idea; it pushes the value zero on the expression stack and then a variable to point to this value:

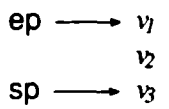
```
stack[++sp].type = INT;
stack[sp].value.integer = 0;
stack[++sp].type = VAR;
stack[sp].value.integer = sp-1;
```

Given that the interpreter is in some expression context, executing the code above has the following affect on the expression stack:

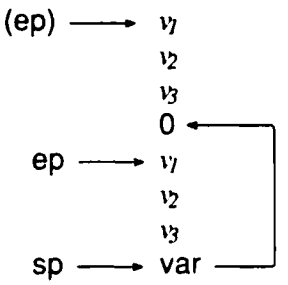


Whenever an instance of repeated alternation is evaluated, the interpreter constructs a variable on the expression stack that is associated with that instance of the control structure. The corresponding **contrep** instruction need only know where that variable is when it gains control. The uniformity of the expression stack structure makes locating this variable straightforward.

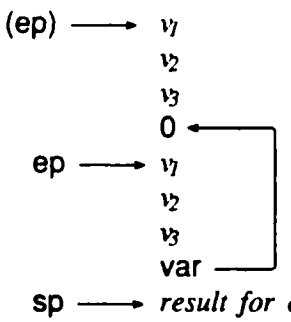
To illustrate this, suppose that the expression stack has the following form when repeated alternation is encountered:



Since repeated alternation is a generator, it follows the general protocol: It saves its failure continuation, copies from the current **ep** to **sp**, and invokes the interpreter with the new context. In addition, just before the copy, **repalt** pushes a value on the stack and after the copy, constructs a variable that points to it. When the interpreter is invoked, the stack has the form:



The repeated alternation expression is evaluated in this new context. There are three possibilities for the expression. It may fail to produce a result, it may produce exactly one result, or it may be a generator and produce many results. If the expression produces exactly one result—that is, if it is not a generator—then the stack has the form:



Notice that the variable is the second descriptor from the top of the stack. On other hand, the expression *expr*$_1$ may be a generator. No matter how complicated this generator is, it follows the protocol for a generator: The context of the stack prior to the evaluation of the generator is copied and the interpreter is called with the **ep** pointing to the new expression context. Thus, after evaluating the generative expression *expr*$_1$, the top of the stack still has the form above: when **contrep** is reached, the repeated alternation variable is the second descriptor from the top

of the stack.

The code for repeated alternation is:

```
case Repalt:
   fipc = ipc;
   ++sp;
   for ( ; ; ) {
      ipc = fipc;
      stack[sp].type = INT;
      stack[sp].value.integer = 0;
      newsp = copy(ep,sp-1);
      stack[++newsp].type = VAR;
      stack[newsp].value.integer = sp;
      signal = interp(sp+1,newsp);
      if (signal != Resume)
         return signal;
      if (stack[sp].value.integer == 0)
         return Resume;
      }
```

The code at the Repalt case pushes the integer 0 on the stack. If the expression succeeds, contrep is reached and this instruction changes the value to 1 to indicate that the expression did succeed. The code for contrep follows:

```
case Contrep:
   stack[stack[sp-1].value.integer].value.integer = 1;
   stack[sp-1] = stack[sp];
   sp--;
   break;
```

Note that besides changing the value of the repeated alternation variable to indicate success, contrep also replaces the variable with the result of the expression and decrements the stack pointer. Otherwise, this variable would interfere with subsequent computations.

2.3.7 Limitation

Unlike repeated alternation, the limitation control structure is not a generator; instead, it limits its generators. In the expression $expr_1 \setminus expr_2$, the task of limitation is to count the results that $expr_1$ produces. When the expression has produced the number of results specified by $expr_2$, the expression is prevented from producing more results. The limitation control structure removes all information relevant to $expr_1$ when it produces its last allowed result. In a sense, limitation can be thought of as providing a generalization of bounded expressions. It generalizes a bounded expression in that it removes the evaluation state of $expr_1$ after it produces its *nth* result—as specified by $expr_2$—rather than after it produces its first result. Unlike the bounded expression, however, limitation does not isolate the evaluation of $expr_1$ from previous contexts. The generated code for $expr_1 \setminus expr_2$ is

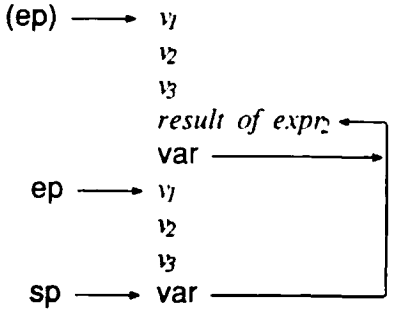
code for $expr_2$
limit
code for $expr_1$
lsusp

In order to control the generation of results for $expr_1$, limit and lsusp must cooperate. The lsusp instruction must cause the expression context to be removed only when the expression produces its last permitted value. The same mechanism for communication used in repeated alternation is used for the limitation control structure. limit pushes a variable on the expression stack to be used by lsusp to count the results produced by the expression. In addition, a new signal is introduced for limitation. The signal Limit is used to the remove the context of the limit expression. Using this signal avoids interference with the Clear signal.

Notice that the value of $expr_2$ is on the stack when limit is reached. limit constructs a variable to point to that value, copies the stack, and establishes a new context by recursing. Also, the variable pointing to the limit value is replicated in order to be available to lsusp. When limit gains control again, its action depends on the signal it receives. If the signal is Clear, limit must return, since the bounded expression in which is occurs is complete. If the signal is Resume, then the expression was not capable of producing the number of results specified by the limit counter and limit propagates the Resume signal. Finally, if the signal is Limit, the expression successfully produced its last allowed result; evaluation continues with the code following limitation. The code in the interpreter for limitation follows:

```
case Limit:
   DeRef(sp);
   if (stack[sp].value.integer == 0)
      return Resume;
   stack[++sp].type = VAR;
   stack[sp].value.integer = sp-1;
   newsp = copy(ep,sp-2);
   stack[++newsp].type = VAR;
   stack[newsp].value.integer = sp-1;
   signal = interp(sp+1,newsp);
   if (signal == Limit && stack[sp].type == INT)
      sp--;
   else
      return signal;
   break;
```

Suppose that arbitrary values v_1, v_2, and v_3 are on the stack in the current context when the limitation control structure is encountered. Then when limit invokes the interpreter, the stack has the form:



Notice that limit constructs two variables pointing to the limit value, one in its current context and one in the new context. When *expr*$_1$ produces its last allowed result, lsusp replaces the limit count descriptor with this result and also replaces the first variable descriptor with the value 0 to indicate that this instance of limitation is complete. Limitation expressions, like repeated alternation, may be nested; the Limit signal must propagate to the appropriate occurrence of limit. The code for lsusp follows:

```
case Lsusp:
   i = stack[sp-1].value.integer;
   if (--stack[i].value.integer > 0) {
      stack[sp-1] = stack[sp];
      sp--;
      }
   else {
      stack[i] = stack[sp];
      stack[i+1].type = INT;
      stack[i+1].value.integer = 0;
      return Limit;
      }
   break;
```

By using the variable pointing to the limit value (now the second descriptor from the top of the stack, since *expr*$_1$ has been evaluated), lsusp decrements the limit value and checks that it is non-zero. If so, *expr*$_1$ has not yet produced all its allowed results. lsusp replaces the variable with the result and execution continues in the current context. If the limit value has reached zero, the current result is the last allowed. lsusp replaces the limit value with this result and replaces the variable just above the limit value with the integer 0. It then returns the Limit signal in order to remove the context of *expr*$_1$.

2.3.8 Iteration

In every *expr*$_1$, the expression *expr*$_1$ is repeatedly resumed until its result sequence is complete. The every expression then fails. Like a bounded expression, the computation of *expr*$_1$ is isolated from previous contexts; thus a mark instruction can be used to begin the evaluation of *expr*$_1$. However, rather than delimiting the code for *expr*$_1$ with an unmark, which removes its context after the first successful evaluation, the expression is delimited with an instruction that forces failure and hence the resumption of *expr*$_1$. The code for every *expr*$_1$ follows:

```
mark0
code for expr1
pop
efail
```

The instruction mark0 does not have a failure continuation operand. Unlike mark, it must transmit failure to a previous context:

```
case Mark0:
   return interp(sp+1,sp);
```

When the efail instruction is reached during execution, it must resume any suspended generators in $expr_1$. Since a Resume signal resumes generators in a previous context, the code for efail is simply:

```
case Efail:
   return Resume;
```

In the expression every $expr_1$ do $expr_2$, for each result in $expr_1$ the expression $expr_2$ is *re-evaluated* in a new goal-directed context. Thus, $expr_2$ is a bounded expression. The virtual machine instructions for every $expr_1$ do $expr_2$ are:

```
mark0
code for expr1
pop
mark0
code for expr2
unmark
efail
```

2.3.9 Break and Next

The break control structure is used to exit from a containing loop expression. The next control structure is used to immediately jump to the control clause of the loop in order to continue evaluation with the next iteration of the loop. In both cases, these control structures must remove expression contexts due to nested bounded expressions before execution continues at the semantically appropriate place. For example, consider the expression

```
while line := read() do
   if find(s,line) then break else write(line)
```

When the break expression is encountered, the bounded expression context due to the do clause must be removed before execution continues after the while expression. For this reason, break and next expressions are not implemented solely by goto instructions. The translator counts the number of nested bounded expressions that are active at the time a break or next is encountered. That number of unmark instructions is emitted before the goto for the control structure. The code for the while expression above is:

```
L1:
        mark0
        var   line
        read
        asgn
        unmark
        mark L1
        mark L3
        var   s
        var   line
        find
        unmark
        unmark          remove bounded do clause
        pnull
        goto  L2        jump to end of loop
        goto  L4
L3:     var   line
        write
L4:     unmark
        goto  L1
L2:
```

The next control structure is similar, but the goto instruction directs execution to the control clause of the loop.

In general, the break control structure may have an argument expression. In that case, the expression is evaluated before the jump to the end of the loop expression. The virtual machine code for break *expr*$_1$ is:

necessary unmark *instructions*
<*code for expr*$_1$>
goto *end-of-loop*

Since the unmark instructions are evaluated before the code for *expr*$_1$, *expr*$_1$ is evaluated in the context in which the loop expression occurs.

2.4 Procedures

As mentioned previously, Icon procedures and local variables are implemented by standard methods. Their implementation is not shown in detail here, but a brief discussion is given to relate their implementation to the recursive interpreter.

The local variables for an Icon procedure are maintained on the expression stack; the variables are referenced through a corresponding pointer maintained across Icon procedure invocations. In keeping with the method of maintaining state information on the C call stack (through recursive calls to the interpreter), a C routine for invoking Icon procedures is used to maintain the activation frames that hold the information related to Icon procedure invocation (for example, the ipc and procedure frame pointer of the calling Icon procedure).

Icon procedures may be generators. A user-defined generator performs the same actions as shown for generative operators and functions: It saves a failure continuation, copies a portion of the expression stack, and recurses to establish a new context. In addition, procedure suspension must also restore the information of the previous Icon procedure held in the current procedure

activation frame.

Bounded expressions can be nested across Icon procedure calls. For example, suppose that words is a procedure that generates the words in a file. This generator can be used as follows:

```
procedure p()
   ...
   if words() == s then ...
   ...
end
```

In p, when the if control clause is evaluated, words is invoked. Evaluation of expressions in words may result in several evaluation contents due to bounded expressions and generators; these contexts may be present when words suspends. If the comparison words() == s succeeds, all of these contexts must be removed. Thus, the Clear signal returned due to the bounded if control clause must clear bounded expressions within words.

This is handled by relating a bounded expression context to the Icon procedure in which the bounded expression occurs. The code in the interpreter for mark is modified to relate its recursive call to interp to the current Icon procedure frame pointer. In the example above, any bounded expression contexts created within words are related to the procedure frame pointer of words. When a Clear signal is returned to the code in the interpreter for mark, the code checks to make sure the current procedure frame pointer has the same value as when the interp call was made. If not, the signal is propagated until it reaches the proper bounding context.

CHAPTER 3

COMPILING EXPRESSIONS FOR ICON

Although compilation techniques for most programming languages are well known, Icon's expression evaluation with generators and goal-directed evaluation presents unusual problems. This chapter presents a model of compilation that is a direct analog of the interpretation techniques given in the previous chapter.

One motivation for developing a compiler is to enable the optimization of expression evaluation. It is possible for a compiler to detect expressions that do not use the full generality of goal-directed evaluation and to improve the code generated for such expressions. The model of compilation presented here is used as a basis for the optimization techniques described in the next chapter.

For pedagogical reasons, the compiler described here generates C code. A production version of the compiler would generate assembly language. Also, C macros are used in the generated code to eliminate unnecessary detail and to focus on the structure of the generated code.

3.1 Overview

The conceptual basis for implementing generators and goal-directed evaluation is the same in the compiled code as in the interpreter. The contexts of evaluation introduced by generators and bounded expressions are maintained and released in the same way. The same two global structures from the interpreter are used in the compiled code: an expression stack for temporary values and an array for global variables. Both of these structures consist of Icon values represented by descriptors as described in Chapter 2. In addition, there is a run-time system supporting the built-in functions and operators of Icon: C functions plus(), numlt(), to(), find(), and so on, one for each function and operation in the language. The functions in the run-time system operate on the temporary values stored on the expression stack. For example, to perform an addition, the plus routine is called and it replaces the top two values on the expression stack with their sum.

Where there is no change of control flow, the correspondence between interpreted actions and compiled code is straightforward. Consider, for example, the expression i + *s and its virtual machine code translation and corresponding compiled code:

```
var i       Pushvar(i_off);
var s       Pushvar(s_off);
size        size(sp);
plus        plus(sp);
            sp--;
```

The compiled code in the right-hand column is a direct translation of the interpreter actions. Note, however, that dereferencing operations, performed by DeRef in the interpreter, do not appear in the compiled code. Dereferencing is performed inside the run-time system routines, in

order to simplify the code presented here. In the compiled code, initial uppercase letters denote a macro and initial lowercase letters denote a function call to a run-time routine. For example, the Pushvar macro pushes a variable on the expression stack and is defined as

```
#define Pushvar(offset) {stack[++sp].type = VAR; \
                         stack[sp].value.integer = offset;}
```

where offset is the constant offset of the variable in iglobals. Other simple virtual machine instructions, such as int, are also implemented as macros. On the other hand, all operations are translated to C function calls. Compare the translation of plus above with that in Chapter 2.

To compile code for generative expressions, bounded expressions, and failure, the interpreter method of handling control flow for generators and goal-directed evaluation is mapped into compiled code. Evaluation contexts are maintained exactly as in the interpreter by generating C functions to maintain the contexts. Each function call in the generated code corresponds to what would have been a recursive call to the interpreter; the body of the function called corresponds to the code that the interpreter would have executed in that context. In that way, evaluation contexts are maintained implicitly by function invocation. Similarly, contexts are removed by returning from calls, either when failure occurs or when the end of a bounded expression is reached.

3.2 The Generated Code

An Icon expression is translated into C functions—as many C functions as are necessary to represent the evaluation contexts of the Icon expression. The generated C functions take the same two arguments as the interpreter routine: the expression pointer and the stack pointer. The compiled code uses the expression stack the same way that the interpreter does: an invocation due to a bounded expression causes the expression pointer to point to the top unused portion of the stack; an invocation due to a generator causes the stack contents from the current expression pointer to be copied, and the expression pointer to point to the base of the replicated values.

The compiled code uses the signals Resume and Clear, returning those signals to drive goal-directed evaluation on failure or to complete the evaluation of a bounded expression, as in the interpreter.

Understanding the compiled code requires understanding how to encapsulate the code for a bounded expression or a generator into a C function, since these are the only points where new contexts are created.

3.2.1 Bounded Expressions

A syntactically bounded expression $expr_1$ is packaged into a separate C function that is invoked when $expr_1$ is to be evaluated, just as the interpreter would have recursed to evaluate $expr_1$ in a new context. The code generator generates arbitrary names for these functions; here, the names are prefixed by context to signify that the function represents a new context for evaluation. The bounded expression $expr_1$ is encapsulated as follows:

```
context1(ep,sp)
int ep, sp;
{
    int signal;

    <code for expr1>
    return Clear;
}
```

The variable signal is not always used in the function. In subsequent examples, it is declared only when necessary.

To illustrate the use of functions, suppose that i < j is a bounded expression of a control structure. The compiled code for the bounded expression is:

```
context1(ep,sp)
int ep, sp;
{
    int signal;

    Pushvar(i_off);
    Pushvar(j_off);
    signal = numlt(sp);
    if (signal  == Resume) return Resume;
    sp--;
    return Clear;
}
```

Since the comparison operation < may fail, the compiled code checks the outcome of the call to numlt and returns Resume if the operation fails.

In compiled code, there is no longer an explicit program counter (ipc in the interpreter). Where evaluation should continue when the function for a bounded expression, such as context1, depends on the control structure that introduced the bounded expression. The flow of control is built into the compiled code at the site where context1 is called, according to the meaning of the control structure.

To illustrate, consider a compound expression:

{ $expr_1$; $expr_2$; $expr_3$ }

The expressions $expr_1$ and $expr_2$ are each bounded. The outcome of $expr_1$ is unimportant; whether it succeeds or fails, evaluation proceeds to $expr_2$. The same is true for $expr_2$; evaluation will proceed to $expr_3$. The generated code for the compound expression is:

```
        .
        .
context1(sp + 1,sp);
context2(sp + 1,sp);
<code for expr3>
        .
        .
```

The bounded expressions $expr_1$ and $expr_2$ are encapsulated as follows:

```
context1(ep,sp)
int ep, sp;
{
   int signal;

   <code for expr1>
   return Clear;
}

context2(ep,sp)
int ep, sp;
{
   int signal;

   <code for expr2>
   return Clear;
}
```

An if expression, on the other hand, checks the outcome of its bounded control clause, as in the code for if *expr1* then *expr2* else *expr3*:

```
                :
                :
   if (context1(sp + 1,sp) == Clear) {
      <code for expr2>
      }
   else {
      <code for expr3>
      }
                :
                :
```

Notice that there is no failure continuation to maintain in the compiled code. The fipc of the interpreter is "compiled out" and is replaced by explicit flow of control.

3.2.2 Generators

In the interpreter, a generator suspends using the following protocol: It copies a portion of the expression stack to preserve temporary values, and recurses to establish a backtracking point and maintain its internal state. Generators do precisely the same thing in the compiled code as well. However, since there is no interpreter to call to establish a new context, a generator is passed an argument that is the function to call in order to continue execution. This argument is known as the *continuation*. The structure of the compiled code for a generator is introduced by first discussing a generative operation. Generative control structures are considered later.

As mentioned previously, an Icon operation has a corresponding C function in the run-time system. In addition to the continuation argument, ep and sp are also passed to the generator so that it can copy that portion of the expression stack when it suspends. The call to a primitive generator has the form:

generator_name(expression pointer, stack pointer, continuation)

The code for to is given below to illustrate how generators are handled in the compiler. Notice that the function for to in the run-time system is nearly identical to the version in the interpreter.

It simply calls its third argument cont when it suspends, whereas the interpreter version recursed:

```
to(ep,sp,cont)
int ep, sp, (*cont)();
{
   int signal, from, limit, newsp;

   DeRef(sp-1);
   DeRef(sp);
   from = stack[sp-1].value.integer;
   limit = stack[sp].value.integer;
   while (from <= limit) {
      newsp = copy(ep,sp-2);
      stack[++newsp].type = INT;
      stack[newsp].value.integer = from;
      signal = (*cont)(sp + 1, newsp);
      if (signal != Resume)
         return signal;
      from++;
      }
   return Resume;
}
```

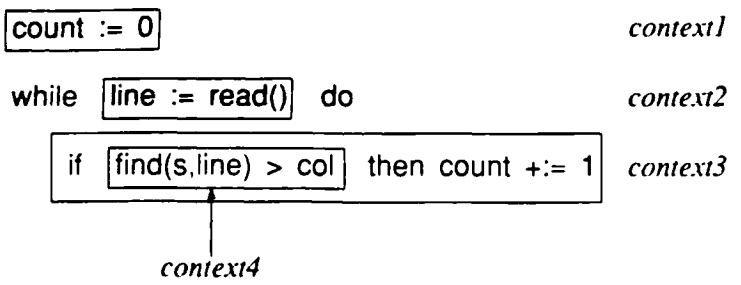
When to suspends, it copies the values on the expression stack from the current ep to the top of stack and calls cont with the new values for ep and sp. It is the compiler's task to properly split an expression containing one or more generators into functions that can be called in the proper sequence to continue evaluation.

3.2.3 An Example

The code generation scheme discussed above can be illustrated with the following example containing nested bounded expressions and a generator:

```
count := 0
while line := read() do
   if find(s,line) > col then count +:= 1
```

These expressions count the number of lines in the input that contain the string s beyond col. The evaluation contexts due to bounded expressions are easily identified, since they correspond to structured syntactic entities. The contexts due to bounded expressions are enclosed in retangles and named below:



The generated code for the assignment and loop expressions is

```
        :
context1(sp+1,sp);
while (context2(sp+1,sp) != Resume)
   context3(sp+1,sp);
        :
```

The traditional control structures of Icon are mapped directly into the corresponding C control structures. The compound expression and while loop of the Icon expression become simple compound and while statements in C, where the control flow is driven by the outcome of evaluating the encapsulated expression contexts. The bodies of context1 and context2 consist of simple monogenic and conditional expressions:

```
context1(ep,sp)
int ep, sp;
{
   Pushvar(count_off);
   Pushint(0);
   asgn(sp);
   sp--;
   return Clear;
}

context2(ep,sp)
int ep, sp;
{
   int signal;

   Pushvar(line_off);
   signal = read(sp);
   if (signal == Resume) return Resume;
   asgn(sp);
   sp--;
   return Clear;
}
```

The body of the while loop is bounded and encapsulated in context3. It in turn contains a bounded expression—the control clause of its if expression, which is encapsulated in context4:

```
context3(ep,sp)
int ep, sp;
{
   int signal;

   signal = context4(sp + 1,sp);
   if (signal == Resume)
      return Resume;
   Pushvar(count_off);
   Dup;
   Pushint(1);
   plus(sp);
   sp--;
   asgn(sp);
   sp--;
   return Clear;
}
```

In context3, the code following the C if statement is the code for count +:= 1. The code for context4 contains the code for the bounded expression find(s,line) > col. Since this expression contains the generator find, context4 is split as follows:

```
context4(ep,sp)
int ep, sp;
{
   Pushvar(s_off);
   Pushvar(line_off);
   return find(ep,sp,context5);
}

context5(ep,sp)
int ep, sp;
{
   int signal;

   Pushvar(col_off);
   signal = numlt(sp);
   if (signal == Resume) return Resume;
   sp--;
   return Clear;
}
```

3.2.4 Alternation

The code generated for alternation is rather straightforward conceptually, and, like the code for any generator, it only appears unusual or difficult because of the code splitting—the method of compiling an expression into several functions. Otherwise, the compiled code for alternation simply mimics the interpreter version.

The alternation expression $expr_1 \mid expr_2$ must first evaluate $expr_1$ and then whatever expression follows alternation. If failure propagates back through $expr_1$ to alternation, then $expr_2$ is

evaluated, and again the code following the alternation expression is evaluated. To do this, the code following alternation must be encapsulated in its own function. In general, the form of the generated code for the alternation expression $expr_1$ | $expr_2$ is:

```
            :
    newsp = copy(ep,sp);
    signal = context1(sp + 1, newsp);
    if (signal != Resume) return signal;
    <code for expr2>
    return context3(ep,sp);
            :

context1(ep,sp)
int ep, sp;
{
    <code for expr1>
    return context3(ep,sp);
}

context3(ep,sp)
int ep, sp;
{
    <code for expression following alternation>
}
```

Notice that context3 contains the code for the expressions following alternation and is called after $expr_1$ is evaluated; if failure propagates back through $expr_1$ to resume the alternation control structure, then $expr_2$ is evaluated and context3 is called again.

The form of the code varies according to the properties of $expr_2$. For example, if $expr_2$ is also a generator, then it is encapsulated into its own function, rather than being emitted directly as shown above.

3.2.5 Repeated Alternation

The code generated for repeated alternation is also similar to the interpreter version. Various parts are encapsulated in functions, but these functions correspond essentially to the interpreter code shown in Chapter 2 for the instructions repalt and contrep. As in the interpreter, repeated alternation pushes the integer 0 on the stack, copies the current context of the expression stack, and then pushes its local control variable on the stack. In the generated code for the expression |$expr_1$, the repeated alternation expression is evaluated in a new context, called context1 in the code below:

```
        :
    ++sp;
    for ( ; ; ) {
        stack[sp].type = INT;
        stack[sp].value.integer = 0;
        newsp = copy(ep,sp-1);
        stack[++newsp].type = VAR;
        stack[newsp].value.integer = sp;
        signal = context1(sp+1,newsp);
        if (signal != Resume) return signal;
        if (stack[sp].value.integer == 0) return Resume;
        }
        :
```

The code generated for context1 varies according to whether *expr*$_1$ is a generator. If *expr*$_1$ is not a generator, the code is as follows:

```
context1(ep,sp)
int ep, sp;
{
    int signal;

    <code for expr1>
    stack[stack[sp-1].value.integer].value.integer = 1;
    stack[sp-1] = stack[sp];
    sp--;
    <expression following repeated alternation>
}
```

As in the interpreter, the value of the repeated alternation control variable is changed to 1 if evaluation of *expr*$_1$ succeeds. The variable descriptor is then replaced with the result of *expr*$_1$ and sp is decremented. Evaluation flows into the code following repeated alternation.

If *expr*$_1$ is a generator, the code in context1 above is split into separate functions. For example, if *expr*$_1$ is 1 to 10 the code is as follows:

```
context1(ep,sp)
int ep, sp;
{
    Pushint(1);
    Pushint(10);
    return to(ep,sp,context2);
}
```

```
context2(ep,sp)
int ep, sp;
{
   int signal;

   stack[stack[sp-1].value.integer].value.integer = 1;
   stack[sp-1] = stack[sp];
   sp--;
   <code for expression following repeated alternation>
}
```

3.2.6 Limitation

The code for $expr_1 \setminus expr_2$ is split into several functions. Again the basic form of the code corresponds to the interpreter version of limitation. In the general case, $expr_1$ is a generator and is encapsulated into a separate function, context1. The code below is almost identical to the code in the interpreter for the limit instruction:

```
<code for expr2>
if (stack[sp].value.integer == 0) return Resume;
stack[++sp].type = VAR;
stack[sp].value.integer = sp-1;
newsp = copy(ep,sp-2);
stack[++newsp].type = VAR;
stack[newsp].value.integer = sp-1;
signal = context1(sp+1,newsp);
if (signal == Limit && stack[sp].value.integer == 0)
   return context3(ep,sp-1);
else
   return signal;
```

After $expr_1$ is evaluated (by calling context1), the limit counter must be checked to insure that $expr_1$ is allowed to produce subsequent results. The code to check the limit counter, which corresponds to lsusp in the interpreter, is in context2 below:

```
context1(ep,sp)
{
   int signal;

   <code for expr1>
   return context2(ep,sp);
}
```

```
context2(ep,sp)
int ep, sp;
{
   int i;

   i = stack[sp-1].value.integer;
   if (--stack[i].value.integer > 0) {
      stack[sp-1] = stack[sp];
      return context3(ep,sp-1);
      }
   else {
      stack[i] = stack[sp];
      stack[i+1] = INT;
      stack[i+1].value.integer = 0;
      return Limit;
      }
}
```

Since $expr_1$ is a generator, it calls the limit checking code, context2, as its continuation when it suspends. Also, the code for the expression following the limitation control structure must be in a separate function, since it can be executed if $expr_1$ is allowed to produce more results and also if $expr_1$ has produced its last result and the Limit signal is returned:

```
context3(ep,sp)
int ep, sp;
{
   int signal;

   <code for expression following limitation>
}
```

3.2.7 Break and Next

Recall from Chapter 2 that break and next require that any contexts due to nested bounded expressions of the enclosing loop be removed. In the interpreter, a sequence of unmark instructions is executed, one for each bounded expression context that is active when the break or next instruction is encountered. This is possible because the interpreter has an explicit program counter, the ipc. That is, consider what happens when two unmark instructions are executed. The interpreter executes the first unmark, all contexts due to the top bounded expression are removed, and the ipc then points to the second unmark instruction. Everything is in order to remove the remaining bounded expression.

Because there is no ipc to manipulate in the compiled code, and the execution paths are "static", the effect of a series of unmark instructions cannot be achieved by the compiled code as easily as in the interpreter. In the compiled code, when the contexts due to a bounded expression and any generators suspended within the bounded expression are removed, execution returns to the point of the function call generated for the bounded expression. To remove the next bounded expression context, another return must be executed. Consequently, a bounded expression within a loop that contains a break or next must know when its context is being removed due to a break or next. For the compiler, two new signals, Break and Next, are

introduced to distinguish the reason for removing a context.

To illustrate this, consider the **while** expression in the example from Section 3.2.3. The example is modified to break from the loop if the string **s** is not found in the input line:

```
count +:= 1
while line := read() do
   if find(s, line) > col then count +:= 1 else break
```

The compiled code for the above expressions is:

```
        :
        :
context1(sp+1,sp);
while (context2(sp+1,sp) != Resume)
   if (context3(sp+1,sp) == Break)
      break;
        :
        :
```

It is the code in **context3**, the do clause, that has been modified and may return the **Break** signal:

```
context3(ep,sp)
int ep, sp;
{
   int signal;

   signal = context4(sp + 1,sp);
   if (signal == Clear) {
      Pushvar(count_off);
      Dup;
      Pushint(1);
      plus(sp);
      sp--;
      asgn(sp);
      sp--;
      return Clear;
      }
   else
      return signal;
}
```

In general, the overhead of checking for the two new signals occurs only in the generated code for contexts in a loop body containing a **break** or **next**.

3.3 Comments on the Code Generator

For the most part, compiling code for Icon is straightforward. The only difficulty in the compiler is breaking the source code into the appropriate functions; this aspect of compilation makes the code generator itself non-trivial (or less straightforward that the translator that generates virtual machine instructions).

The main problem is that the code generated for an expression depends on the code surrounding the expression. For example, consider the expression

(*expr*$_1$, if *expr*$_2$ then *expr*$_3$ else *expr*$_4$, *expr*$_5$)

Whether some or all of *expr*$_3$, *expr*$_4$, and *expr*$_5$ need to go into separate C functions depends on whether they contain generators. If those expressions are monogenic, then the generated code contains a C function call only for the bounded if control clause, *expr*$_2$. The code is:

```
<code for expr1>
signal = context2(ep,sp);
if (signal == Clear) {
   <code for expr3>
   }
else {
   <code for expr4>
   }
<code for expr5>
```

However, if *expr*$_4$ is a generator, then it must call the code for *expr*$_5$, which means that *expr*$_5$ must be encapsulated in a separate function. This function is then used to continue evaluation after the selected expression of the if is evaluated. For example, if *expr*$_4$ is the expression find(s,line), the the generated code is

```
        .
        .
        .
<code for expr1>
signal = context2(ep,sp);
if (signal == Clear) {
   <code for expr3>
   return context5(ep,sp);
   }
else {
   Pushvar(s_off);
   Pushvar(line_off);
   return find(ep,sp,context5);
   }
        .
        .
        .
context5(ep,sp)
int ep, sp;
{
      int signal;

      <code for expr5>
}
```

Similarly, the actual code generated for alternation depends on the expressions surrounding the alternation control structure, and likewise for other control structures.

The code generator can easily employ a simple code generation scheme: It can encapsulate every expression within its own function. This is wasteful, however, both in terms of the size of the generated code and the execution speed of the resulting compiled code. Generating the code described in this chapter requires performing case analysis on the components of syntactic constructs to determine whether they actually require to be generated in separate functions. Conceptually, this can be thought of as an optimization of the simple code generation scheme.

CHAPTER 4

OPTIMIZING EXPRESSION EVALUATION

The evaluation mechanism of Icon is general and powerful. Not all programs, however, make use of the general capabilities of goal-directed evaluation. For example, an Icon program typically contains many expressions that are not generators, and even many expressions that contain generators never make use of backtracking. Compile-time semantic analysis can gather information about the properties of expressions and how they are used at their lexical sites. This information can then be used by a compiler in order to generate more efficient code for expression evaluation.

The discussion here is concerned with the optimization of control flow aspects of generators and goal-directed evaluation. Other language features, such as untyped variables, are not considered here, even though they influence the efficiency of the implementation.

4.1 Unnecessary Bounding

Expressions are bounded in order to limit the scope of goal-directed evaluation. Bounding avoids backtracking into an expression whose evaluation is semantically complete. In that sense, bounding controls the lifetime of expressions. For example, in

if *expr*$_1$ then *expr*$_2$ else *expr*$_3$

once *expr*$_1$ is evaluated, subsequent failure in either *expr*$_2$ or *expr*$_3$ must not cause backtracking into *expr*$_1$. In the virtual machine code, a bounded expression is delimited by mark/unmark pairs. The operational effect of bounding is to remove any state related to the bounded expression once its lifetime is complete. Of course, only a bounded expression containing one or more generators may have a non-trivial evaluation state, due to suspension, that must be removed. Many expressions that are syntactically bounded, such as the control clause of an if expression, may not actually contain generators. For example, in

if i < count then *expr*$_2$ else *expr*$_3$

the control clause is a simple conditional expression. Actual bounding of the control clause during evaluation is unnecessary: After the control clause is evaluated, there is no suspended state, and therefore nothing to be erroneously resumed if either *expr*$_2$ or *expr*$_3$ should fail.

Icon programs typically consist of a mixture of expressions that require full goal-directed evaluation and expressions that do not. An example is the procedure locate below. It writes out the lines of input that contain the string s and returns a count of such lines.

```
procedure locate(s)
   local count, linenum, line

   count := 0
   linenum := 0
   while line := read() do  {
      linenum +:= 1
      if find(s,line) then {
           write(lineno, ": ", line)
           count +:= 1
           }
      }
   if count > 0 then return count else fail

end
```

There are ten syntactically bounded expressions in locate, yet only the control clause of the first if expression contains a generator. None of the remaining syntactically bounded expressions requires actual bounding during evaluation. Indeed, it is shown in the next section that even the if control clause in the example above does not require the general evaluation strategy.

4.2 Unnecessary Suspension

The results of a generator are produced as required by its lexical context, with failure causing the generator to be resumed for its next result. Consider the expression

if i < upto(c,s) then $expr_1$ else $expr_2$

In an attempt to make the comparison succeed, expression evaluation may resume upto many times; specifically, upto will be resumed each time the comparison operation < fails. In a bounded expression, when a generator occurs together with conditional expressions that may resume the generator, it usually cannot be determined *a priori* how many results will be required from the generator by the surrounding lexical context.

On the other hand, generators do occur in situations where it can be determined statically that at most one result is required from the generator. This happens when a generator is being used as a simple monogenic or conditional expression. In such a case, the generator is not required to generate a sequence of values; it produces its first result and suspends, but is never resumed during its lifetime. For example, consider the expression

i := upto(c,s);

The upto expression may fail initially, in which case the assignment is not made and the entire expression fails. However, if upto produces a result, the assignment is made and evaluation of the assignment expression is complete. Since the entire expression i := upto(c,s) is bounded, once the assignment is made, the lifetime of the expression is complete. Once upto suspends with its result, there is nothing in the surrounding context that fails, and hence nothing that may resume upto.

Since a generator suspends so that it can be resumed later if needed, suspension is useless when failure cannot occur during the lifetime of the generator. Such a generator can be "demoted" to a simple conditional or monogenic expression; that is, it can be implemented in a non-retentive way. For example, a generator can be called with an argument that indicates

whether it should suspend or simply return after computing its result.

Demoted generators increase the number of bounded expressions that do not need to be actually bounded, as discussed in the previous section. For example, in the if expression from locate in the previous section

```
if find(s,line) then ...
```

the control clause contains a generator that is not resumed during its lifetime. Therefore, this invocation of find can be demoted to a conditional. As a result, no expression in locate requires the general goal-directed evaluation mechanism.

For this discussion, the term generator is somewhat misleading. There are some "generators" in Icon that produce at most one result—the function tab is an example. This function does not produce subsequent results when it is resumed, but rather restores state related to string scanning. It uses suspension as a means to gain control when failure occurs, not as a means to provide further results. For this reason, it would be more accurate to use the term *retentive expression* rather than generator, since the implementation issue is that an expression might require state retention, regardless of whether it produces many results. From this point on in the discussion, the term generator should be identified with the term retention expression.

4.3 Properties of Expressions

In order to avoid unnecessary bounding and unnecessary suspending, the following information must be known at compile time:

1. For each generator, is the generator in the path of failure? Or put another way, is the generator resumable during its lifetime?
2. For each bounded expression, does it contain resumable generators?

A simple attribute grammar, given in Section 4.4, can be used to describe the above attributes of bounded expression and generators, how they relate to each other, and how their lexical context affects them. Attribute grammars are described in [2, 43].

4.3.1 Resumption

The semantics of Icon segregate the program into the separate, isolated syntactic units introduced by the bounded expressions of control structures. In the absence of break, which is discussed below, there is no dynamic interaction between the bounded expressions in the program. For example, in

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

the characteristics of $expr_2$, $expr_3$, and the expression in which the if appears, do not affect $expr_1$. The converse is true also.

Bounding localizes the resumption paths, since resumption paths of separate bounded expressions are disjoint. Within a bounded expression, an expression is resumable if failure can reach the expression during its lifetime. Consider the expression

$$(expr_1\ op_1\ expr_2)\ op_2\ expr_3$$

where op_1 and op_2 are arbitrary operations. It is convenient to write the expression in postfix form in order to understand the behavior of failure:

((*expr*$_1$, *expr*$_2$) *op*$_1$, *expr*$_3$) *op*$_2$

Expressions are evaluated from left to right and resumed from right to left. Referring to the postfix form of the expression, an expression that fails can cause the resumption of anything to its left. If *op*$_1$ fails, it can resume *expr*$_1$ or *expr*$_2$. Conversely, whether or not a given expression can be resumed depends on whether conditional expressions occur in subsequent evaluation. Thus, *expr*$_2$ may be resumed during evaluation if any of the operations or expressions to its right are conditional—*op*$_1$, *expr*$_3$, and *op*$_2$ are each possible resumers of *expr*$_2$. Notice, however, that *expr*$_1$ cannot possibly resume *expr*$_2$, nor anything else to its right.

The expression upto(c, !a) < 4 provides a concrete example. In postfix form, the expression is

((c, !a) upto, 4) <

The potential resumers of the second argument of upto (the expression !a) are upto, 4, and <. Since the constant expression 4 is monogenic, only upto and < can possibly resume !a.

4.3.2 Control Structures

All control structures are expressions. There are two things to keep in mind for expressions that are control structures. A control structure introduces a new evaluation state for those expressions that it bounds, and it also has some effect on the context in which it occurs. For example, in

x := if *expr*$_1$ then *expr*$_2$

the if expression is conditional: if *expr*$_1$ fails, the if expression fails, since there is no else clause. Its control clause, on the other hand, is evaluated in a separate context. Likewise in

(*expr*$_1$, if *expr*$_2$ then *expr*$_3$ else *expr*$_4$, *expr*$_5$)

the if control structure introduces a new evaluation state for its control clause *expr*$_2$, but it affects the outer expression according to the properties of its then and else clauses. Thus, *expr*$_1$ is resumable if any of *expr*$_3$, *expr*$_4$, or *expr*$_5$ can fail.

Almost all control structures may fail. For example, all the looping control structures fail when the iteration is complete. In general, a control structure introduces failure into its surrounding context.

The break control structure introduces an exception, since it may have an argument expression. A loop can be exited with break *expr*$_1$; the outcome of the loop expression is the outcome of *expr*$_1$. For example, in

```
while expr1 do {
   ...
   if expr2 then break upto(c,s)
   ...
   }
```

the loop expression either fails or exits with the generative upto expression. In this case, the loop expression must be treated as a generative expression. To summarize, in the absence of break, all looping control structures are conditional; if break *expr*$_1$ occurs in the body of a loop, the properties of the *expr*$_1$ are taken into account in order to properly treat the loop control structure.

4.4 The Attribute Grammar

Several attributes are used to collect information needed to optimize the code generated for expression evaluation. The attributes indicate if an expression can fail, if it is resumable, and if it contains suspending expressions.

Recall that in postfix form, an expression is resumable if an expression to its right can fail. When an expression is represented by an abstract syntax tree, the relationship between a node and its possible resumers is as follows: a node is resumable if its parent can resume it, or if a right sibling can resume it. Therefore, resumption information comes from both above and below in an abstract syntax tree. Inherited and synthesized attributes are used to relate this information. As described in [2], a synthesized attribute is defined in terms of attributes in children nodes. An inherited attribute is defined in terms of attributes in parent and/or sibling nodes.

Two attributes are required to relate information concerning failure and resumption. The information inherited from the parent is kept in the attribute resume. The information synthesized from the children is kept in the attribute fail. A third attribute, suspend, indicates whether resumable generators occur in a subtree. For most productions, the suspend attribute is defined only in terms of attributes from children nodes. However, in some cases, suspend is defined in terms of resume. The suspend attribute is therefore inherited. Two additional attributes are required to handle break expressions within loops. These attributes are introduced later, after the fundamental ideas have been introduced through the simpler productions. Initially, the discussion assumes that break expressions are not present.

The attributes suspend, resume, and fail are boolean-valued; the functions representing attribute computation use logical *or*. For example, in a production representing an addition expression

expr → *expr*$_1$ + *expr*$_2$
  expr.suspend = *expr*$_1$.suspend ∨ *expr*$_2$.suspend

the attribute suspend is defined as the logical *or* of the children's suspend attributes.

The use of the attributes is explained below in the productions for Icon expressions. There is one general rule for the assignment of values to the attributes: A bounded expression begins a new context of evaluation in which, initially, there is no resumption. Resumption is introduced only as nodes representing conditional expressions are encountered in the syntax tree for the bounded expression.

Since the body of a procedure is bounded, evaluation always takes place in a bounded expression. A simplified production for a procedure is:

procedure → *declarations expr* end
  expr.resume = false
  procedure.suspend = *expr*.suspend
  procedure.fail = *expr*.fail

The inherited attribute resume is assigned false, indicating that the expression for the procedure body is not resumable initially. The productions for expressions are given below.

Constants and identifiers are monogenic expressions and need only set the values of suspend and fail:

expr → *literal*
expr.suspend = false
expr.fail = false

expr → *identifier*
expr.suspend = false
expr.fail = false

In the production for an operator, the definitions of the attributes depend on the properties of the specific operators. Consider the production for a binary operator:

expr → *expr*$_1$ binop *expr*$_2$

Whether the operands are in a resumable context depends both on the value inherited through *expr*.resume and on the properties of the specific operation. For example, the binary operation + cannot resume its operands, but < may. It simplifies the explanation to categorize the operations according to their intrinsic properties, such as whether an operation is monogenic, conditional, or generative, as described in Appendix B. These categories disguise some information, however. For example, all the generators listed in Appendix B are also conditional. To make this information explicit, the following categories are used: unconditional/nonretentive (monogenic), unconditional/retentive, conditional/nonretentive (conditional), and conditional/retentive (generative). Version 6 of Icon does not contain operations in the second category, but since an unconditional generator could easily be added to the language, this category is included here for completeness. The terms within parentheses relate the categories to those listed in Appendix B. The productions for binary operations are given below. Other operations are treated similarly.

Case 1: unconditional/nonretentive. This type of operation cannot resume its children. It passes down the current value of resume to its right child. Whether the left child is resumable depends on both the value of resume and the value of fail from its right sibling. In addition, a nonretentive operation does not suspend even if the context is resumable. The entire expression suspends or fails according to how these attributes are defined by the children:

expr → *expr*$_1$ *un_binop* *expr*$_2$
expr$_2$.resume = *expr*.resume
expr$_1$.resume = *expr*.resume ∨ *expr*$_2$.fail
expr.suspend = *expr*$_1$.suspend ∨ *expr*$_2$.suspend
expr.fail = *expr*$_1$.fail ∨ *expr*$_2$.fail

Case 2: conditional/nonretentive. A conditional operation may resume its children. It disregards the inherited value of resume and indicates a resumable context by assigning the inherited attribute true for both its children. It also sets fail to true, since the operation is conditional. Since operations in this category are nonretentive, suspend is defined as in Case 1:

expr → *expr*$_1$ *cn_binop* *expr*$_2$
expr$_2$.resume = true
expr$_1$.resume = true
expr.suspend = *expr*$_1$.suspend ∨ *expr*$_2$.suspend
expr.fail = true

Case 3: unconditional/retentive. The children's resume attributes and also the fail attribute are handled as in Case 1, since this operation is unconditional. A retentive operation suspends,

however, if the context inherited from the parent is resumable; for this reason, the definition of suspend relies on resume, as well as the suspend attributes of its children:

expr → *expr*$_1$ *ur_binop* *expr*$_2$
  expr$_2$.resume = *expr*.resume
  expr$_1$.resume = *expr*.resume ∨ *expr*$_2$.fail
  expr.suspend = *expr*.resume ∨ *expr*$_1$.suspend ∨ *expr*$_2$.suspend
  expr.fail = *expr*$_1$.fail ∨ *expr*$_2$.fail

Case 4: conditional/retentive. A conditional retentive operation may fail and so its children are resumable. The suspend attribute is defined as in Case 3, since the operation is retentive:

expr → *expr*$_1$ *cr_binop* *expr*$_2$
  expr$_2$.resume = true
  expr$_1$.resume = true
  expr.suspend = *expr*.resume ∨ *expr*$_1$.suspend ∨ *expr*$_2$.suspend
  expr.fail = true

A control structure may consist of both expressions that are syntactically bounded and not bounded. For the expressions that are bounded, the context for evaluation begins afresh with resume set to false. The unbounded expressions relate to the outer context.

The if expression fails if either of its children from the then or else clauses fails, and suspends if either suspends:

expr → if *expr*$_1$ then *expr*$_2$ else *expr*$_3$
  expr$_1$.resume = false
  expr$_2$.resume = *expr*.resume
  expr$_3$.resume = *expr*.resume
  expr.suspend = *expr*$_2$.suspend ∨ *expr*$_3$.suspend
  expr.fail = *expr*$_2$.fail ∨ *expr*$_3$.fail

Likewise, a compound expression gets the synthesized values of suspend and fail from its second, unbounded expression and begins a new context of evaluation for its first expression:

expr → *expr*$_1$; *expr*$_2$
  expr$_1$.resume = false
  expr$_2$.resume = *expr*.resume
  expr.suspend = *expr*$_2$.suspend
  expr.fail = *expr*$_2$.fail

Some control structures bound all of their argument expressions; the control structure itself is a conditional expression and does not suspend. For example, not *expr*$_1$ has only one argument and it is bounded. Even if *expr*$_1$ could suspend, since it is bounded, the control expression not *expr*$_1$ does not suspend. Likewise, loop expressions are conditional expressions (in the absence of break). Thus,

expr → not *expr*$_1$
  expr$_1$.resume = false
  expr.suspend = false
  expr.fail = true

expr → while *expr*$_1$ do *expr*$_2$
expr$_1$.resume = false
expr$_2$.resume = false
expr.suspend = false
expr.fail = true

expr → until *expr*$_1$ do *expr*$_2$
expr$_1$.resume = false
expr$_2$.resume = false
expr.suspend = false
expr.fail = true

expr → repeat *expr*$_1$
expr$_1$.resume = false
expr.suspend = false
expr.fail = true

The every control structure differs from the other forms of iteration since the control clause expression is not bounded, but rather is resumed for each iteration of the loop. Its first argument expression, then, is always resumable, but its second argument, which is bounded, is not:

expr → every *expr*$_1$ do *expr*$_2$
expr$_1$.resume = true
expr$_2$.resume = false
expr.suspend = false
expr.fail = true

Alternation has state retention and suspends if the inherited context is resumable, or if its arguments suspend:

expr → *expr*$_1$ | *expr*$_2$
expr$_1$.resume = *expr*.resume
expr$_2$.resume = *expr*.resume
expr.suspend = *expr*.resume ∨ *expr*$_1$.suspend ∨ *expr*$_2$.suspend
expr.fail = *expr*$_1$.fail ∨ *expr*$_2$.fail

Repeated alternation is like alternation. It is a retentive control structure and it suspends if the inherited context is resumable, or if its argument suspends:

expr → |*expr*$_1$
expr$_1$.resume = *expr*.resume
expr.suspend = *expr*.resume ∨ *expr*$_1$.suspend
expr.fail = *expr*$_1$.fail

The arguments to limitation are resumable if the context is resumable; it simply passes down the resume attribute. Likewise, the control structure suspends or fails if either of its children suspends or fails:

expr → *expr*$_1$ \ *expr*$_2$
  expr$_1$.resume = *expr*.resume
  expr$_2$.resume = *expr*.resume
  expr.suspend = *expr*$_1$.suspend ∨ *expr*$_2$.suspend
  expr.fail = *expr*$_1$.fail ∨ *expr*$_2$.fail

The break control structure complicates attribute computation, which so far has been discussed without consideration of this control structure. For example, the break expression can be used as follows:

```
while ... do {
   repeat {
      ...
      break break expr1
      ...
      }
   }
```

When the break expression of the repeat loop is evaluated, both loops are exited. The effect is that the while loop is replaced with *expr*$_1$. In terms of attribute computation, this means that *expr*$_1$ must have access to the resume attribute of the containing while loop, since *expr*$_1$ is resumable if failure can propagate to the while loop. Furthermore, if *expr*$_1$ can suspend, this information must be integrated with the suspend attribute of the while loop. Since several break expressions may occur within a loop, the suspend attributes of all break expressions related to a given loop must be known to the loop expression.

Two additional attributes are needed to relate the information of loop control structures and break expressions. To introduce the new attributes, consider initially a simple situation where loops are not nested. Let loopresume be an inherited attribute used to relate the resumption context of the loop expression to *expr*$_1$ in break *expr*$_1$, and let brksusp be a synthesized attribute that propagates up to the enclosing loop expression the information concerning whether *expr*$_1$ suspends. Using the production for a while expression to illustrate this, the relationship between these new attributes and the existing resume and suspend attributes is:

expr → while *expr*$_1$ do *expr*$_2$
  expr$_1$.loopresume = *expr*.resume
  expr$_2$.loopresume = *expr*.resume
  expr.suspend = *expr*$_1$.brksusp ∨ *expr*$_2$.brksusp

expr → break *expr*$_1$
  expr$_1$.resume = *expr*.loopresume
  expr.brksusp = *expr*$_1$.suspend

Since brksusp is a simple synthesized attribute (like fail), productions representing terminal symbols initialize it by defining it to be false. All productions must propagate loopresume to argument expressions. Likewise, since any of the expressions in a production may contain a break expression, these expressions may contribute to the possible suspension of the containing loop. Thus, in all productions the brksusp attributes of argument expressions are merged. The production for a compound expression illustrates how the new attributes are handled:

expr → *expr*$_1$; *expr*$_2$
  expr$_1$.loopresume = *expr*.loopresume
  expr$_2$.loopresume = *expr*.loopresume
  expr.brksusp = *expr*$_1$.brksusp ∨ *expr*$_2$.brksusp

The relationship between loopresume and brksusp as described above is straightforward. In general, however, loops may be nested and break expressions may consist of further break expressions, as in the example at the beginning of this discussion. Nested loops and multiple breaks act as begin-end pairs. Consequently, the attributes loopresume and brksusp must be stacks of boolean values rather than simple boolean values. Each production for a loop control structure must push its resume attribute on the loopresume stack. At a given point in parsing, the depth of loopresume corresponds to the current depth of the nested loop expressions. In the production for break *expr*$_1$, the top value of the loopresume stack is used as the resume attribute for *expr*$_1$. It also pushes its expression's suspend attribute on the brksusp stack.

In an attribute grammar, all computations on attributes must be applicative. In the following productions, the attributes loopresume and brksusp are defined as lists that are manipulated as stacks by the usual applicative operations defined for lists: cons, first, and rest.

A production that represents a terminal symbol, such as a literal, defines brksusp as the empty list. As before, all productions must propagate loopresume and brksusp. Note that since the brksusp attribute is a list, its definition at the production for a compound expression, given above, is no longer correct. Since each expression's brksusp attribute is a list, the two lists must be merged. The production for a compound expression, defining all attributes, is:

expr → *expr*$_1$; *expr*$_2$
  expr$_1$.resume = false
  expr$_2$.resume = *expr*.resume
  expr$_1$.loopresume = *expr*.loopresume
  expr$_2$.loopresume = *expr*.loopresume
  expr.suspend = *expr*$_2$.suspend
  expr.fail = *expr*$_2$.fail
  expr.brksusp = merge(*expr*$_1$.brksusp,*expr*$_2$.brksusp)

The function merge takes two lists and returns a list whose elements are a pairwise logical *or* of the two argument lists. If the two lists are unequal in length, the shorter list is appended with false values to make them the same size. This is necessary, since loops do not necessarily have associated break expressions.

The modified production for while shows how the stack-valued attributes are properly handled in the general case:

expr → while *expr*$_1$ do *expr*$_2$
  expr$_1$.resume = false
  expr$_2$.resume = false
  expr$_1$.loopresume = cons(*expr*.resume, *expr*.loopresume)
  expr$_2$.loopresume = cons(*expr*.resume, *expr*.loopresume)
  expr.suspend = first(*expr*$_1$.brksusp) ∨ first(*expr*$_2$.brksusp)
  expr.fail = true
  expr.brksusp = merge(rest(*expr*$_1$.brksusp),rest(*expr*$_2$.brksusp))

A loop production uses the top value of the brksusp stack, which indicates whether any

enclosing break expressions suspend. A loop may not have associated break expressions, in which case brksusp is an empty list. To handle this, the function first is defined to return false when given an empty list. Note that a loop production must also propagate the remaining stack of values brksusp from its children.

The production for break is:

expr → break $expr_1$
  $expr_1$.resume = first(*expr*.loopresume)
  $expr_1$.loopresume = rest(*expr*.loopresume)
  expr.suspend = false
  expr.fail = false
  expr.brksusp = cons($expr_1$.suspend,$expr_1$.brksusp)

Note that break effectively pops loopresume before propagating it to $expr_1$, since break expressions in $expr_1$ relate to the next containing loop.

The attribute definitions are non-circular. An informal argument is the following:

(i) The attribute fail is a synthesized attribute and can be evaluated in one bottom-up pass.

(ii) The resume and loopresume attributes can then be evaluated in one top-down pass.

(iii) The suspend and brksusp attributes can then be evaluated in one bottom-up pass.

Attribute evaluation requires only one pass, however, because of the semantics of failure propagation in Icon. Expressions are evaluated left-to-right and resumed from right-to-left. Consequently, as shown previously in the productions for operations, the fail attribute of a right-sibling affects the resume attribute inherited by a left-sibling. The definition of resume in Case 1 is an example:

$expr_1$.resume = *expr*.resume ∨ $expr_2$.fail

This relationship is always right-to-left: fail attributes of left-siblings do not affect right-siblings. During attribute evaluation, if nodes in the parse tree are visited in preorder reversed fashion, that is, *parent*, *rightmost-child*, *left-most child*, attribute evaluation can be accomplished in one top-down pass.

4.5 Application to code generation

Code is generated from the annotated trees produced after parsing. The code generator inspects the values of the attributes stored in the nodes of the tree for an expression in order to generate optimized code. The following sections discuss how the information from the annotated parse trees is used in the compiled code model of Chapter 3.

4.5.1 Bounded Expressions

The compiler model uses a call/return scheme in the generated code to maintain and use information about suspended generators and active bounded expressions. If bounding is not necessary for a given bounded expression, there are two effects on the generated code. First, the bounded expression is not encapsulated in a separate function. Second, operations in the bounded expression that can fail do not return; rather, the failure continuation is handled by explicit flow of control to a compile-time destination. The code generator either uses a C control structure to direct control flow, or it uses gotos. For example, the expression if i < count then $expr_1$ else $expr_2$ is compiled as follows:

```
        :
Pushvar(i_off);
Pushvar(count_off);
signal = numIt(sp);
if (signal == Resume) then {
   sp--;
   <code for expr2>
   }
else
   {
   sp--;
   <code for expr3>
   }
        :
```

On the other hand, a while loop whose argument expressions need not be bounded would be translated using gotos. For example, the code generated for while i < limit do *expr1* is

```
l1:
        Pushvar(i_off);
        Pushvar(limit_off);
        signal = numIt(sp);
        if (signal == Resume) { sp -= 2; goto l2; }
        sp--;
        <code for expr1>
        goto l1;

l2:
```

4.5.2 Generative Operations

In the compiler, a generative function or operation is called differently depending on whether the generator is resumed during its lifetime, which is indicated by the resume attribute. A new argument is added to a generator; this argument tells the generator to either return or to suspend after it produces its result. For example, the to operation is called in one of two ways:

```
to(ep, sp, context3, Suspend)
```

or

```
to(ep, sp, NoCont, Return)
```

The first form of the call indicates that to must suspend with its result and continue evaluation by calling context3. The second form indicates that to need not suspend, but should simply return after pushing its result on the expression stack. Each of the run-time routines for a generative operation or function is modified to suspend only when necessary. The modified code for to illustrates this:

```
to(ep,sp,cont,action)
int ep, sp, (*cont)(), action;
{
   int signal, from, limit, newsp;

   DeRef(sp-1);
   DeRef(sp);
   from = stack[sp-1].value.integer;
   limit = stack[sp].value.integer;
   while (from <= limit) {
      stack[sp-1].type = INT;
      stack[sp-1].value.integer = from;
      if (action == Return)
           return Return;
      newsp = copy(ep,sp-1);
      signal = (*cont)(sp + 1, newsp);
      if (signal != Resume)
         return signal;
      from++;
      }
   return Resume;
}
```

Referring again to locate from Section 4.1, recall that no expression of this procedure requires general evaluation:

```
procedure locate(s)
   local count, linenum, line

   count := 0
   linenum := 0
   while line := read() do  {
      linenum +:= 1
      if find(s,line) then {
           write(lineno, ": ", line)
           count +:= 1
           }
      }
   if count > 0 then return count else fail

end
```

The optimized code for the procedure body is:

```
        Pushvar(count_off);         /* count := 0 */
        Pushint(0);
        asgn(sp);
        sp--;
        Pushvar(linenum_off);       /* linenum := 0 */
        Pushint(0);
        asgn(sp);
        sp--;
l1:     Pushvar(line_off);          /* while line := read() ... */
        Pushvar(read_off);
        signal = read(sp);
        if (signal == Resume) {
            sp -= 2;
            goto l2;
            }
        asgn(sp);
        sp--;
        Pushvar(linenum_off);       /* linenum +:= 1 */
        Dup;
        Pushint(1);
        plus(0);
        sp--;
        asgn(sp);
        sp--;

        Pushvar(s_off);             /* if find(s,line) */
        Pushvar(line_off);
        signal = find(ep,sp,NoCont,Return);
        sp--;
        if (signal != Resume) {     /* then        */
            Pushvar(lineno_off);
            Pushstr(": ");
            Pushvar(line_off);
            write(sp,3);
            sp -= 3;
            Pushvar(count_off);
            Dup;
            Pushint(1);
            plus(sp);
            sp--;
            asgn(sp);
            sp--;
            }
        sp--;
        goto l1;
l2:
```

4.5.3 Alternation

Alternation is often used in contexts where only one result is expected. In this case, alternation is being used as a branch control structure, such as in:

```
(f := open("inputfile")) | stop("can't open file")
```

The resume attribute at the alternation node in the parse tree is false in such an instance. The code generator can emit a simple branch statement. The code for the alternation expression above is simply

```
        Pushvar(f_off);
        Pushstr("inputfile");
        signal = open(sp,1);
        if (signal == Resume) {
                sp -= 2;
                goto l1;
                }
        asgn(sp);
        sp -= 1;
l1:
        Pushstr("can't open file");
        stop(sp);
```

4.6 Application of Optimizations in Previous Implementations

A compiler can avoid bounding more often than the interpreter presented in Chapter 2. In the interpreter, bounded expressions that are strictly monogenic can be translated without the mark/unmark, avoiding bounding. But this is not true for bounded expressions that contain conditional expressions. For example, in

```
if x < y then expr1
```

the virtual machine instruction for the comparison would have to be extended to include the failure label. All conditional operations and functions would require two forms of the instruction. In general, this proliferation of instructions could make the interpreter unacceptably large.

Previous implementors have noted the inefficiency of bounding expressions that are strictly conditional or monogenic, but only the Version 2 implementation made attempts to avoid unnecessary bounding. Hanson and Korb [28] noted that monogenic expressions do not require bounding (mark/unmark in later versions) and they avoided bounding for these expressions. However, bounded expressions that contain conditional expressions still require bounding in their implementation model.

CHAPTER 5

CONCLUSIONS

5.1 Performance

The recursive interpreter presented in Chapter 2 is fully implemented and operational. The actual implementation of the recursive interpreter interfaces with the Icon Version 6 run-time system. Version 6 of Icon, which is written in C, has been publicly distributed and is in wide use. In [22], the representation of data, storage management, and other details of the full implementation of Version 6 of Icon are described.

The semantic analysis phase and the code generator of the compiler are written in Icon. The compiler generates C code that also interfaces with a Version 6 run-time system, with only simple modifications to account for properties of the generated functions, such as the number and type of arguments passed to the run-time routines, as mentioned in Section 4.5.2.

Using recursion as the basis of implementation significantly reduces the complexity of implementing expression evaluation inherent in previous implementations. It is useful as a conceptual model and as a tool for understanding the operation of expression evaluation. Performance is not the main concern for the recursive interpreter; however, measurements show that the recursive model does not incur significant performance penalties. For the compiler, some performance measurements are given in order to estimate the benefits of compiling Icon programs.

5.1.1 Interpreter

To evaluate the effect of recursion on the performance of the interpreter, there are two main issues to consider: the cost of recursive calls in terms of time and the amount of stack space needed. It is difficult to compare the recursive interpreter with all of its predecessors in detail, since their performance depends on many matters that are not related to the issues here. However, some valid comparisons can be made between the recursive interpreter and the Version 6 interpreter.

Both the recursive interpreter and the Version 6 interpreter are written in C. They are structurally similar, except for the more general use of recursion in the recursive interpreter in place of the explicit construction of frames on the interpreter stack in Version 6. Comparing these two interpreters raises the question of the comparative cost of the two approaches in handling information for expression evaluation.

Timing tests show no measurable difference in running speed between the Version 6 interpreter and the recursive interpreter on a wide range of programs, although it is possible to contrive programs that favor one or the other interpreter.

There is, however, a difference in stack use. While there are very substantial variations from program to program, the average high-water mark on the system stack, which is used for C calls, is about four times greater for the recursive interpreter than for the Version 6 interpreter. On the

other hand, the average high-water mark on the expression stack for the recursive interpreter is about one-half that of the Version 6 interpreter.

For computers with a small amount of memory, the amount of system stack used by the recursive interpreter could limit the kinds of programs it could handle. However, the amount of system stack used by the recursive interpreter for suspended generators is limited by the number of generators that are suspended at any one time. In Icon, this number typically is relatively small; a maximum of five is typical.

5.1.2 Compiler

Several features of Icon incur a heavy run-time burden. Examples include untyped variables, polymorphic operations, and very high-level built-in features such as string scanning and set and table lookup. The compiler uses a run-time system that is similar to the one in Version 6, and the control flow optimizations do not affect the time spent in the run-time system. The gains in execution speed apply only to the portion of the total execution time directly involved in expression evaluation. Profiling the Version 6 interpreter on a large suite of programs suggests that, on the average, programs spend approximately 20% of their total execution time in the interpreter itself and the remaining portion is spent in the run-time system. For both the recursive interpreter and the Version 6 interpreter, execution time spent in the interpreter proper is that time spent for decoding instructions and implementing expression evaluation. The optimizations discussed in Chapter 4 can reduce only that portion of execution time.

Difference in execution speed between interpreted program and compiled programs, without optimizations, is due to the fact that the instruction decoding loop is omitted. In the absence of optimization, expression stack usage is the same for the compiler and recursive interpreter. With the optimizations, many of the invocations for maintaining expression evaluation contexts are eliminated, as a result of removing unnecessary bounding and demoting generators.

To illustrate the affects of compilation and expression evaluation optimization, several programs, ranging in size and style, are evaluated below. Appendix C contains source listings of the programs; they are briefly summarized here.

- meander computes a meandering string. A string over an alphabet of k symbols is said to be n-meandering if every word of length n is contained in the string. The meandering string computed has length k^n. This program illustrates the use of strings, and also typical combinations of control structures.
- power computes exponentiation (x^y) in arbitrary-precision arithmetic. It illustrates how Icon control structures are used to advantage in a conventional problem.
- queens solves the well-known problem of placing eight queens in nonattacking positions on a chess board. It illustrates the use of generators and backtracking in problem that is suited to combinatorial search.
- roman converts Arabic numbers to Roman numbers. Although small, it is included to demonstrate that programs often rely on the powerful built-in functions of Icon, in this case the function map.
- rsg generates randomly selected sentences from a grammatical specification. It illustrates extensive use of lists, tables, string scanning, and generators in a nontrivial program.

- tournament uses a heuristic method to assign partners for four-handed bridge. It illustrates how generators are typically used in combination with data structures.

The figures for unnecessary bounding and demoted generators for these programs follow:

program	*lexically bounded expressions*	*number eliminated*	*generators*	*generators demoted*
meander	16	13	6	3
power	70	62	14	4
queens	16	11	6	0
roman	12	11	4	3
rsg	150	131	59	20
tournament	106	83	32	5

These figures are difficult to evaluate in isolation. For execution speed, their importance depends on the frequency of execution of the corresponding contexts that are eliminated. In terms of stack space utilization, the figures represent a direct effect on the amount of system stack space used for recursion, and expression stack space used for replicated temporary values. When bounding and generating are reduced in compiled programs, the system and expression stack high-water marks decrease accordingly. In addition, the compiler is able to generate compact code when bounding is eliminated.

The following performance measurements are based on comparing the total running times of the programs when compiled against the total running times of the programs when interpreted with the recursive interpreter. When compiled with optimizations, the overall performance of these programs improves anywhere from 4% to 32%. The table below shows the percentage increase in execution speed for three situations: compiled programs versus interpreted programs (*c vs ri*), compiled programs with optimizations versus compiled programs without optimizations (*c_o vs c*), and the overall comparison of compiled programs with optimizations versus interpreted programs (*c_o vs ri*).

program	*c vs ri*	*c_o vs c*	*c_o vs ri*
meander	6%	2%	8%
power	21%	8%	29%
queens	18%	5%	23%
roman	24%	8%	32%
rsg	13%	8%	21%
tournament	3%	1%	4%

Since Icon programs spend a significant portion of execution time in the run-time system, which is unaffected by this model of compilation and the corresponding optimizations, it is worthwhile to estimate how much time the test programs spend only in the interpreter for expression evaluation and thereby estimate how compiling and optimizing programs is affecting the time spent for expression evaluation. The Version 6 interpreter has been profiled using the utility Gprof [13] in order to estimate the portion of time spent inside the interpreter (the time spent for expression evaluation proper), and the portion of time spent in the run-time system. The run-time systems of the recursive interpreter and the Version 6 interpreter are the same, and the test programs perform the same under each interpreter. The following table shows the time spent for expression evaluation for each of the test programs. Using this information gives a rough estimate

of how time spent for expression evaluation is affected by compiling and performing the optimizations:

program	*time spent for expression evaluation*	*compiler + opts improvement*	*portion of expression evaluation time reduced*
meander	13%	8%	62%
power	35%	29%	82%
queens	34%	23%	67%
roman	35%	32%	91%
rsg	25%	21%	84%
tournament	13%	4%	30%

The methods discussed here can affect only a fixed portion of the execution time. Put in that light, the test programs show that the techniques are successful at eliminating cost of expression evaluation. Notice that the programs power, roman, and rsg show the best performance improvements. In these programs, many bounded expression and generator contexts are eliminated by the optimizations.

For some programs, simply removing bounding is significant. For example, in queens, one procedure (place) is called 15,720 times. The body of place is an if expression whose control clause need not be bounded—a significant savings, given the number of times the control clause is evaluated.

The tournament program shows that compiling and optimizing programs as discussed in the dissertation cannot significantly improve the performance of all programs. tournament makes extensive use of the set data type of Icon, and subsequently spends most of its execution time in the run-time system performing operations on sets. Furthermore, this program has several procedures that are called repeatedly. However, most of these procedures contain every expressions with generators—a combination that cannot be improved by the optimizations. For example, the procedures select and remove dominate the execution of tournament. The select procedure is:

```
procedure select(s1,base,setting)
   local s2
   if s2 := member(zero[s1],!base) then {
      every delete(zero[!setting],s2)
      every delete(zero[s2],!setting)
      }
   else if s2 := member(one[s1],!base) then {
      every delete(one[!setting],s2)
      every delete(one[s2],!setting)
      }
   else fail
   return s2
end
```

select is called 360 times, and never fails; thus, two every expressions are evaluated 360 times (each time select succeeds). Likewise, remove is called 480 times and evaluates generators each time.

The optimizations have a significant impact on the size of the generated code. The following

table shows the reduction in the size of the code for the optimized compiled code when compared to the unoptimized compiled code:

program	*code size reduction*
meander	32%
power	33%
queens	18%
roman	40%
rsg	31%
tournament	31%

The overall speed-up for optimized, compiled programs versus interpreted programs is modest, but using these techniques reduces a large portion of the execution time that otherwise is required for expression evaluation. The optimizations performed for generators and bounded expressions stand to be more significant when combined with optimizations geared towards other aspects of Icon, such as type inference, that affect time spent in the run-time system. Reducing the run-time system overhead would increase the impact of expression evaluation optimization proportionately. In terms of space, the optimizations currently incur a significant savings. For programs like tournament, which do not run significantly faster when compiled, the reduced code size makes including the optimizations worthwhile nonetheless.

5.2 Related Work

The method of compiling Icon expression evaluation described in Chapter 4, based on transforming the interpreter model into a compiler design, can be thought of as performing a partial evaluation of the interpreter upon a given program text. The compiler symbolically executes the Icon program text, generating C code that would have been executed by the interpreter and linking generated modules by function calls. The C code can then be fully evaluated given the program data. The theory of partial evaluation is discussed in [11]. Recently, research efforts have explored the pragmatics of using partial evaluation as the basis of developing compilers from interpreters [35], and even as a means to automatically generate a compiler generator [26].

Although optimization of expression evaluation for Icon has not previously been studied, some of the concepts are similar to optimization techniques in other languages with backtracking. In particular, a great deal of work has been done in the area of optimizing Prolog programs. Several papers discuss finding determinant predicates [33, 47], a notion generalized in [9]; also work in semi-intelligent backtracking attempts to optimize control flow [5]. Whether for Icon or Prolog, the optimizations are similar in that they attempt to avoid the expense of a backtracking point (a suspended generator in Icon terms) when it is known that the backtracking point is not useful. At the conceptual level, the intent of the optimizations is the same. However, since the languages differ so widely in their semantics, the semantic information used to find and perform the optimizations is different.

There are other possible methods for implementing generators. For example, [24] discusses how to implement coroutines in Scheme by using continuations. Since generators are a restricted form of coroutines, continuations can also be used to implement generators. Continuations were first introduced as a mathematical formalism to describe the semantics of programming languages; continuation-style compilation of Icon would most likely resemble its denotational description [23]. Since continuations can represent the denotational semantics directly, such an

implementation offers a different point of view from the operational methods described in the dissertation.

It is difficult to accurately evaluate an alternate method—such as a continuation model—without actually implementing it. However, one observation is that the continuation method may use more space due to the lifetime properties for continuations: The continuations used to represent evaluation contexts would exist throughout program execution unless removed by a garbage collection. Extensive work has been done in Scheme compilation to infer how continuations are used and thereby to implement them more efficiently [30]. However, it appears that the behavior of continuations modeling Icon goal-directed evaluation falls outside their optimization techniques. In contrast, in the recursive model, contexts are deallocated when a bounded expression's evaluation is complete, since they are represented by simple functions with stack behavior.

5.3 Future Work

5.3.1 Language Considerations

The benefits of optimizing compiled code vary with programming style and problem domain. Icon programs range in the degree to which they use goal-directed evaluation and generators, both because of the programmer's writing style and the inherent structure of the problem being solved. Programs using string scanning or making exclusive use of generators and goal-directed evaluation do not benefit as much as more conventional programs from the optimizations considered in the dissertation, since these are exactly the general forms of expression evaluation that cannot be reduced by the optimizations. However, if the optimizations do not apply to a given program, this does not mean that the program is inherently inefficient. Indeed, the opposite is true. Programs that do use the full capabilities of Icon evaluation can be more efficient than those that do the same thing conventionally, since generators and goal-directed evaluation internalize computation that otherwise would have to be written out explicitly.

Consider, for example, the eight-queens problem. In Icon, a programmer may use the goal-directed evaluation style, as shown in queens, or may program in a conventional style. Appendix C contains an iterative version of eight-queens written in Icon (queens2). Comparing the execution times of these two programs shows that the iterative version of eight-queens is slower that the goal-directed version, both when interpreted and compiled. Although this is only one example, it illustrates the point that generators and goal-directed evaluation provide a convenient notation for otherwise tedious computation; writing out the computation explicitly does not result in better performance.

Goal-directed evaluation is not inherently inefficient; what appears to be inefficient is a completely general implementation of goal-directed evaluation in a language that also allows conventional computation to be expressed directly.

It is important for future optimization work to know how Icon programs are used in practice. Information about the density of generators in programs and the use of goal-directed evaluation would help to determine where emphasis should be place when developing optimization techniques. To this end, the analysis could be modified to give additional information about issues such as how often generators are used in implicit goal-directed contexts versus how often they are used in conjunction with every; the common programming idioms formed by built-in generators and control structures; and combinations of generative control structures that occur frequently. This kind of information is extremely useful for the language design/implementation cycle.

Such information might indicate whether is it worthwhile to enhance performance not only through compile-time analysis and optimization, but also by providing additional language features as well. For example, it may be worthwhile to add new control structures that are a composition of existing control structures, but that may be implemented more efficiently than the present ones. Consider the following composition of repeated alternation and limitation that often occurs in Icon programs:

(|*expr*$_1$) \ *expr*$_2$

This could be formulated with a new control structure:

expr$_1$! *expr*$_2$

This control structure would be able to implement the semantics of combining repeated alternation and limitation much more efficiently. In the first form, the repeated alternation control structure reevaluates *expr*$_1$ in an infinite loop (with a check for failure of *expr*$_1$); the containing limitation control structure then signals termination when it reaches its limit. Using the alternative form, the loop termination for ! would be combined with the limit on the result sequence as specificed by *expr*$_2$, eliminating the nesting done previously.

It is also worth noting that other generative control structures have been suggested for Icon and related languages. Some of these are:

- Forward alternation [10, 27]: This control structure can actually be expressed in Icon as (*expr*$_1$ | *expr*$_2$) \ 1. Information gathered as described above might tell about the importance of optimizing this combination.
- exclusive alternation: Let ! denote exclusive alternation. Then the result sequence for *expr*$_1$! *expr*$_2$ is the result sequence for *expr*$_1$, if it is not empty, and is the result sequence for *expr*$_2$ otherwise. This control structure cannot be formulated in Icon using existing control structures, since the formulation if *expr*$_1$ then *expr*$_1$ else *expr*$_2$ is incorrect if *expr*$_1$ has side affects.
- invocation limitation [45]: This control structure limits the invocation a function or operation to one result, without limiting the arguments. This cannot be formulated in Icon using existing control structures. For example, in find(!a,s) \ 1, the entire expression is limited to one result. If find produces a result, goal-directed evaluation does not resume !a. In the compiler, invocation limitation already is handled in the implementation by the ability to call generators to produce at most one result. To provide this at the source level only requires introducing syntax for the control structure.

Whether additional control structures, such as those described above, are beneficial enough to justify including them in the language is an open question. Examining the current practices of Icon programmers would be quite useful in answering such a question.

The information about the properties of expressions that is used for optimization can also be useful for other applications. One example is the detection of programming errors due to unexpected failure. Consider the following expression:

```
i := find(s1,s2)
```

Since find can fail, the assignment may never take place. Such coding may be intentional, or it may be an instance of careless programming where the programmer does not consider that i may not be assigned a value. In addition to unexpected failure, analysis can also give information about

ambiguous failure, as in

```
while n := integer(read()) do ...
```

Again, the programmer may expect the loop to terminate only when the the end of input is reached, but the explicit type-conversion function can also fail. The conversion should take place separately so that erroneous input can be detected. The formulation of the loop is incorrect if it is expected to terminate only when the end of input is reached. Since the analysis already identifies potential failure within bounded expressions, it would be a simple matter to adapt it to include a finer-grained attribute evaluation to detect such situations. The compiler might then be used in a mode where it provides such information to the user, with or without actually compiling the program. Such information could be used to aid the programmer in formulating correct Icon programs.

5.3.2 Optimization

There are several possibilities for further optimization, ranging from detailed, specialized optimizations applicable to specific control structures, to larger, pervasive issues.

One possibility is to transform common, idiomatic uses of the every control structure into straightforward code that uses a loop in place of generation. Although every may involve complicated expressions and side-effects, it is often used to express iteration over a simple result sequence. For example, a typical use of every has the following form:

```
every i := 1 to *s do {
   <expressions using i>
   }
```

Such occurrences could be mapped to a while loop, either through a source-to-source mapping, or source-to-target language mapping. For example, a source-to-source transformation could map the above every expression into

```
i := 1
temp := *s
while i <= temp do {
   <expressions using i>
   i +:= 1
   }
```

The variable temp is a unique local introduced by the mapping. This kind of optimization is similar in spirit to that described in Chapter 4—removing the unnecessary generality of evaluation due to generators and goal-directed evaluation—but it probably is best performed by an Icon source-to-source transformation. Such an optimization resembles, in nature, tail-recursion optimization in Scheme [39]. Note, however, that in Scheme the recursion is due to source-level function recursion, whereas in Icon, the recursion is due to the implementation of generators and backtracking.

As described in Chapter 4, some generators are not in the path of failure and therefore need not suspend. There are also many cases where a generator, although in the path of failure, cannot generate subsequent results that satisfy the expressions that may resume it. For example, consider the following expression:

```
tab(find(s1,s2))
```

The result sequence for find—regardless of the values of s1 and s2—consists of a sequence of zero or more increasing integers. If tab fails, it is not possible for find to produce a result that can cause tab to subsequently succeed, using the same arguments. (If find's arguments are generative expressions, then this is not true, since find could be eventually be invoked with a new set of arguments.) There are many cases where knowledge about the impossibility of success is useful, particularly in string scanning, where the behavior of the built-in generators used in the expressions is well-defined.

A variant of this is illustrated by the expression from the procedure remove in tournament in Appendix C:

```
s1[find(s2,s1)] := ""
```

Since subscripting is a conditional operation and find is an argument to subscripting, the attribute computation of Chapter 4 concludes that find is resumable; it is compiled as a suspending generator. However, find, if it succeeds, produces a proper index of s1. Subscripting s1 by this index cannot fail, by definition. In this instance, since the subscripting does not fail, find is never resumed. To detect that find cannot be resumed in this situation, it is necessary to use knowledge about the expression in which find is embedded.

To whatever extent further optimization of control flow patterns and specific control structures pays off in reduced execution time, it still does not affect the place where most the time is spent during execution—the run-time system. Type checking, type conversion, and the many high-level operations in the language that are performed in the run-time system take up a large part of the execution time of typical Icon programs. In particular, untyped variables incur a large cost during execution time. In the absence of type information, all primitive operations and functions must be structured to handle the most general case of operands, and type checking is performed whether necessary or not. A type inference system for Icon is necessary to achieve significant performance improvements for compiled programs. Type inference for Icon has been studied by Walker [44]. His results indicate that Icon variables are typically used in a type-consistent way: over a wide range of programs, approximately 90% of operands can be inferred by his type inference system. Using this information, the generated code could be specialized to perform type checking when necessary and omit it when not. In addition, given knowledge about operand types, some of the run-time operations could be reduced to a small amount of code and could therefore be generated in-line, avoiding a run-time function call altogether. Combining type information and control flow information during the code generation process would result in much more efficient code than is now possible. This is the first step in reducing the overhead of the run-time system.

5.3.3 Code Generation

A production version compiler for Icon would generate assembly language rather than C. The generated C code presented in Chapter 3 is very regular and essentially provides a template for the structure of corresponding assembly language. The code that manipulates values on the expression stack is easily transformed to assembly language instructions. The remaining code consists of functions calls, tests, and gotos. The function calls can be translated into assembly language instructions that interface properly with the function call protocol of the C compiler for the host machine (assuming that the run-time system remains written in C).

5.4 Retrospective

Using recursion in an interpreter is hardly new [1, 32]. In a language such as Lisp, recursion in an interpreter follows naturally from a traversal of a tree representation of the source program. Similarly, the resolution algorithm for logic programming languages such as Prolog is also expressed concisely by recursion [7, 8, 41], where the recursion corresponds to a depth-first traversal of the SLD-tree of the program. However, the computational basis of a logic programming language is quite different from that of Icon. Prolog programs consist of declarative statements and there is no provision for controlling the computation of the program. In Icon, on the other hand, both the traditional control structures and the novel control structures for generators are an important aspect of the linguistic facilities of the language.

The use of recursion for handling expression evaluation in Icon is important because it provides a conceptually clear approach to handling generators and goal-directed evaluation in an imperative language. Recursive calls isolate evaluation in new contexts and provide a natural mechanism for saving state information. It is easy to implement new control structures because of the correspondence between new evaluation contexts and the recursion used in their implementation. Using the recursive model, evaluation of Icon expressions can be described operationally as a process of recursing at failure continuation points and returning, either to resume generators or discard them.

Performance is not a significant issue for the recursive interpreter. Its advantages are its conceptual simplicity and its possible application to extensions to Icon and the implementation of similar types of expression evaluation in other programming languages such as C [6] and Pascal [12]. While details vary for different languages and implementation frameworks, the same principles apply.

Applying the recursive model to compilation of Icon programs gives an immediate working ground to explore and experiment with optimization of expression evaluation. The optimizations discussed here focus on the fundamental aspects of goal-directed evaluation. They attempt to reduce the general implementation of goal-directed evaluation to a conventional form in those cases where the full generality of Icon's expression evaluation mechanism is not used. It appears that these optimizations help to reduce execution time that is used to maintain expression evaluation information.

Icon provides powerful mechanisms for formulating many complex programming operations in concise and natural ways. However, generators, goal-directed evaluation, and related control structures introduce implementation problems that do not exist for languages with only conventional expression evaluation. This dissertation presents an implementation model using recursion that can be used for both an interpreter and a compiler. In the case of the compiler, optimizations can be performed to improve the efficiency of Icon programs, mainly by reducing the general evaluation strategy whenever possible.

There remain many possibilities for extending this work. Perhaps the most promising is its potential integration with optimization of other aspects of the Icon programming language, such as type checking and conversion.

Acknowledgments

I would like to thank my advisor, Ralph E. Griswold, for his support and encouragement throughout my years as a student. I have enjoyed and been influenced by his broad knowledge in the field of computer science and his sense of aesthetics in programming languages. The underlying ideas in the dissertation have been inspired by his thoughts.

I thank David R. Hanson, now at Princeton University, for his willingness to contribute to this work over a long distance. His extensive, insightful, and prompt comments on drafts of the dissertation have helped to improve both the content and presentation of the work. Dave Hanson's energy is boundless. Conversations with him were always a source of motivation and enthusiasm.

I am indebted to Peter J. Downey for his careful reading of the dissertation, and for providing many criticisms and helpful comments related to this work.

I also thank Richard D. Schlichting, who has provided help and encouragement in many ways over the years.

Many of my fellow graduate students have supported me, generously giving of their time to discuss this work with me. They have also helped to create a comfortable and interesting working atmosphere. I would like to thank Dave Gudeman, Tom Hicks, John Kececioglu, Bill Mitchell, Kelvin Nilsen, Titus Purdin, Ken Walker, and Alan Wendt.

And finally, I thank my friends and family for their support, their warmth, and their humor.

Appendix A: Bounded Expressions

An Icon control structure implicity bounds one or more of its argument expressions, according to the semantics of the language. In the following list of control structures, a bounded expression is enclosed in a rectangle.

Compound Expressions

{ $\boxed{expr_1}$; $\boxed{expr_2}$; ... ; $expr_n$ }

Selection Expressions

if $\boxed{expr_1}$ then $expr_2$ else $expr_3$

case $\boxed{expr_1}$ of {

$\boxed{expr_2}$: $expr_3$

$\boxed{expr_4}$: $expr_5$

⋮

default: $expr_n$

}

Negation

not $\boxed{expr}$

Loops

repeat [*expr*]

while [$expr_1$] do [$expr_2$]

until [$expr_1$] do [$expr_2$]

every $expr_1$ do [$expr_2$]

Procedures

procedure p()

[*expr*]

end

return [*expr*]

suspend *expr*

Conjunction

$expr_1$ & $expr_2$ & ... & $expr_n$

Repeated Alternation

|$expr_1$

Limitation

$\boxed{expr_1}$ \ $expr_2$

Note that limitation bounds $expr_1$ according to the number of results specified by $expr_2$.

String Scanning

$expr_1$? $expr_2$

Appendix B: Operations and Functions

In Icon, some operations and functions are similar to those in traditional languages and always produce a value. Others may fail to produce values, or may produce many values. The operations and functions of Icon are classified in this appendix as follows: a monogenic expression produces exactly one value; a conditional expression may produce a value, or may fail to produce a value; a generator may produce a sequence of zero or more values. See [18] for a complete description of the meanings of the operations and functions.

Operations

Monogenic	Conditional	Generative
~*expr*	*expr*	!*expr*
−*expr*	/*expr*	=*expr*
+*expr*	?*expr*	
^*expr*		
**expr*		
.*expr*		
expr$_1$ || *expr*$_2$	*expr*$_1$ === *expr*$_2$	*expr*$_1$ <- *expr*$_2$
expr$_1$ — *expr*$_2$	*expr*$_1$ == *expr*$_2$	*expr*$_1$ <-> *expr*$_2$
expr$_1$ / *expr*$_2$	*expr*$_1$ >>= *expr*$_2$	
expr$_1$ ** *expr*$_2$	*expr*$_1$ >> *expr*$_2$	
expr$_1$ ||| *expr*$_2$	*expr*$_1$ <<= *expr*$_2$	
expr$_1$ − *expr*$_2$	*expr*$_1$ << *expr*$_2$	
expr$_1$ % *expr*$_2$	*expr*$_1$ ~== *expr*$_2$	
expr$_1$ * *expr*$_2$	*expr*$_1$ ~=== *expr*$_2$	
expr$_1$ + *expr*$_2$	*expr*$_1$ = *expr*$_2$	
expr$_1$ ^ *expr*$_2$	*expr*$_1$ >= *expr*$_2$	
expr$_1$ ++ *expr*$_2$	*expr*$_1$ > *expr*$_2$	
	expr$_1$ <= *expr*$_2$	
	expr$_1$ < *expr*$_2$	
	expr$_1$ ~= *expr*$_2$	
	expr$_1$:= *expr*$_2$	
	expr$_1$:=: *expr*$_2$	
	expr$_1$[*expr*$_2$]	
	expr$_1$[*expr*$_2$:*expr*$_3$]	*expr*$_1$ to *expr*$_2$ by *expr*$_3$

Functions

Monogenic	Conditional	Generative
abs	any	bal
center	cset	find
close	get	move
collect	integer	seq
copy	any	tab
delete	match	upto
display	member	
exit	numeric	
image	open	
insert	pop	
left	pos	
list	proc	
map	pull	
push	read	
put	reads	
repl	real	
reverse	string	
right		
set		
sort		
stop		
system		
table		
trim		
type		
write		
writes		

Appendix C: Icon Programs

The programs whose performance measurements are discussed in the dissertation are shown in this appendix.

```
#
#          M E A N D E R I N G   S T R I N G S
#
#
#  This main procedure accepts specifications for meandering strings
#  from standard input with the alphabet separated from the length by
#  a colon.
#

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

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

```
#
#                    P O W E R
#
#
# This program computes a ^ b in arbitrary-precision arithmetic.
# Author: Paul Abrahams.
# Main program is modified to compute one instance of a^b repeatedly.

procedure main()
    local a, b, bbits, prod, time
    i := 1
    while i < 500 do {
        a := integer(37)
        b := integer(15)
        bbits := binrep(b)
        if prod := power(a,bbits) then
            print_it(prod)
        else
            write("Too many digits in the result!")
        i +:= 1
        }
end

# Compute the binary representation of n (as a string)

procedure binrep(n)
    local retval
    retval := ""
    while n > 0 do {
        retval := n % 2 || retval
        n /:= 2
        }
    return retval
end

# Compute a to the power bbits, where bbits is a bit string.
# The result is a list of coefficients for the polynomial a(i)*k^i.

procedure power(a, bbits)
    local b, m1, retval
    m1 := (if a >= 10000 then [a % 10000, a / 10000] else [a])
    retval := [1]
    every b := !bbits do {
        (retval := product(retval, retval)) | fail
        if b == "1" then
            (retval := product(retval, m1)) | fail
        }
    return retval
end
```

```
# Compute a*b as a polynomial in the same form as for power.

procedure product(a,b)
    local i, j, k, retval, x
    if *a + *b > 5001 then
        fail
    retval := list(*a + *b, 0)
    every i := 1 to *a do
        every j := 1 to *b do {
            k := i + j - 1
            retval[k] +:= a[i] * b[j]
            while (x := retval[k]) >= 10000 do {
                retval[k + 1] +:= x / 10000
                retval[k] %:= 10000
                k +:= 1
            }   }
    every i := *retval to 1 by -1 do
        if retval[i] > 0 then
            return retval[1+:i]
    return retval[1+:i]
end

procedure print_it(n)
    local ds, i, j, k
    ds := ""
    every k := *n to 1 by -1 do
        ds ||:= right(n[k], 4, "0")
    ds ?:= (tab(many("0")), tab(0))
    ds := repl("0", 4 - (*ds - 1) % 5) || ds

    every i := 1 to *ds by 50 do {
        k := *ds > i + 45 | *ds
        every j := i to k by 5 do {
                        ds
           writes(ds[j+:5], " ")
                        }
        write()
        }
    write()
end
```

```
#
#       E I G H T  -  Q U E E N S (1)
#
#
#  This program prints the possible ways to place eight queens
#  on a chess board in non-attacking positions.
#

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

procedure q(c)
   suspend place(1 to 8, c)     # look for a row
end

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

procedure bwrite(s)
   local i, l
   every r := 1 to 8 do {
      l := repl(" .",8)
      s[i:=(1 to 8)] = r & (l[2*i] := "Q")
      write (l)
      }
   write()
end
```

```
#
#          R O M A N   N U M E R A L S
#
#
#  This program takes Arabic numerals from standard input and writes
#  the corresponding Roman numerals to standard outout.
#

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

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

```
#
#       R A N D O M   S E N T E N C E   G E N E R A T I O N
#
#
#       This program generates randomly selected strings ("sen-
#   tences") from a grammar specified by the user.  Grammars are
#   basically context-free and resemble BNF in form.
#   See [21] for a description of the input specifications.
#

global defs, ifile, in, limit, prompt, tswitch

procedure main(args)
   local line, plist, s, opts, time

                                          # procedures to try on input lines
   plist := [define, generate, grammar, source, comment, prompter, error]
   defs := table()                        # table of definitions
   defs["lb"] := [["<"]]                  # built-in definitions
   defs["rb"] := [[">"]]
   defs["vb"] := [["|"]]
   defs["nl"] := [["\n"]]
   defs[""] := [[""]]

   opts := getopt(args, "tl+l+")[1]
   limit := \opts["l"] | 1000
   tswitch := \opts["t"]
   &random := \opts["s"]

   ifile := [&input]                      # stack of input files
   prompt := ""
   while in := pop(ifile) do {            # process all files
      repeat {
         if *prompt ~= 0 then writes(prompt)
         line := read(in) | break
         while line[-1] == "\\" do line := line[1:-1] || read(in) | break
         (!plist)(line)
         }
      close(in)
      }
end
```

```
#  getopt(arg, optstring) -- Get command line options.
#
#  This procedure analyzes  the -options on the command line invoking an
#  Icon program.  The argument arg is the argument list as passed to the
#  main procedure, and optstring is the string of allowable option letters.

procedure getopt(arg, optstring)
   local x, i, c, otab, flist, o, p
   /optstring := string(&lcase ++ &ucase)
   otab := table()
   flist := []
   while x := get(arg) do
      x ? {
         if ="-"  & not pos(0) then
            while c := move(1) do
               if i := find(c, optstring) + 1 then
                  otab[c] :=
                    if any(':+.', o := optstring[i]) then {
                       p := "" ~== tab(0) | get(arg) |
                          stop("No parameter following ", x)
                       case o of {
                          ":": p
                          "+": integer(p) |
                                stop("-", c, " needs numeric parameter")
                          ".": real(p) |
                                stop("-", c, " needs numeric parameter")
                          }
                      }
                     else 1
               else stop("Unrecognized option: ", x)
         else put(flist, x)
      }
   return [otab, flist]
end

#  process alternatives

procedure alts(defn)
   local alist
   alist := []
   defn ? while put(alist, syms(tab(upto('|') | 0))) do move(1) | break
   return alist
end

#  look for comment

procedure comment(line)
   if line[1] == "#" then return
end
```

```
#  look for definition

procedure define(line)
   return line ?
      defs[2(="<", tab(find(">::=")))] := 2(move(4), alts(tab(0)))
end

#  note erroneous input line

procedure error(line)
   write("*** erroneous line:  ", line)
   return
end

#  generate sentences

procedure gener(goal)
   local pending, symbol
   pending := [[(goal)]]
   while symbol := get(pending) do {
      if \tswitch then
         write(&errout, symimage(symbol), listimage(pending))
      case type(symbol) of {
         "string":   writes(symbol)
         "list":  {
            pending := ?\defs[symbol[1]] ||| pending | {
               write(&errout, "*** undefined nonterminal:  <", symbol[1], ">")
               break
               }
            if *pending > \limit then {
               write(&errout, "*** excessive symbols remaining")
               break
               }
            }
         }
      }
   write()
end
```

```
#  look for generation specification

procedure generate(line)
   local goal, count
   if line ? {
      ="<" &
      goal := tab(upto('>')) \ 1 &
      move(1) &
      count := (pos(0) & 1) | integer(tab(0))
      }
   then {
      every 1 to count do
         gener(goal)
      return
      }
   else fail
end

#  get right hand side of production

procedure getrhs(a)
   local rhs
   rhs := ""
   every rhs ||:= listimage(!a) || "|"
   return rhs[1:-1]
end
```

```
#  look for request to write out grammar

procedure grammar(line)
   local file, out, name
   if line ? {
      name := tab(find("->")) &
      move(2) &
      file := tab(0) &
      out := if *file = 0 then &output else {
         open(file,"w") | {
            write(&errout,"*** cannot open ",file)
            fail
            }
         }
      }
   then {
      (*name = 0) | (name[1] == "<" & name[-1] == ">") | fail
      pwrite(name,out)
      if *file ~= 0 then close(out)
      return
      }
   else fail
end

#  produce image of list of grammar symbols

procedure listimage(a)
   local s, x
   s := ""
   every x := !a do
      s ||:= symimage(x)
   return s
end

#  look for new prompt symbol

procedure prompter(line)
   if line[1] == "=" then {
      prompt := line[2:0]
      return
      }
end
```

```
#  write out grammar

procedure pwrite(name,ofile)
   local nt, a
   static builtin
   initial builtin := ["lb","rb","vb","nl","","&lcase","&ucase","&digit"]
   if *name = 0 then {
      a := sort(defs,3)
      while nt := get(a) do {
         if nt == !builtin then {
            get(a)
            next
            }
         write(ofile,"<",nt,">::=",getrhs(get(a)))
         }
      }
   else write(ofile,name,"::=",getrhs(\defs[name[2:-1]])) |
      write("*** undefined nonterminal:  ",name)
end

#  look for file with input

procedure source(line)
   local file
   return line ? (="@" & push(ifile,in) & {
      in := open(file := tab(0)) | {
         write(&errout,"*** cannot open ",file)
         fail
         }
      })
end

#  produce string image of grammar symbol

procedure symimage(x)
   return case type(x) of {
      "string":   x
      "list":  "<" || x[1] || ">"
      }
end
```

```
#  process the symbols in an alternative

procedure syms(alt)
   local slist
   static nonbrack
   initial nonbrack := ~'<'
   slist := []
   alt ? while put(slist, tab(many(nonbrack)) |
      [(2(="<", tab(upto('>')), move(1)))])
   return slist
end

#  stop, noting incorrect usage

procedure Usage()
   stop("usage:  [-t] [-l n] [-s n]")
end
```

```
#
#                    T O U R N A M E N T
#
#   Compute partners for four-handed bridge, trying to avoid
#   duplicate pairings.
#

global zero, one, letters

procedure main(a)
   every 1 to 3 do {
   pair := table(0)
   zero := table()
   one:= table()
   number := a[1] | 20
   &random := a[2]
   write("&random=", &random)
   letters := &lcase || &ucase
   labels := letters[1+:number]              # Label the players with letters.
   every c := !labels do                     # Set things up.
      zero[c] := singles(labels, c)
   every c := !labels do
      one[c] := singles(labels, c)
   every round := 1 to 8 do {                # Go through each round.
      write("\n", "round ", round, ":", "\n")
      players := shuffle(labels)             # Mix up the players.
      every 1 to number / 4 do {             # Divide into tables.
         setting := s1 := ?players           # Pick one at random
         players := remove(players, s1)      # and remove from the pool.
         write("s1=", s1)
         until *setting = 4 do {             # Attempt a setting.
            s1 := select(s1, players, setting) | stop("cannot construct")
            setting ||:= s1                  # Add on a player.
            write("setting=", setting)
            players := remove(players, s1)
            }
         displays(setting)                   # Display the result.
         aa := []
         every push(aa,1(s := string(!setting ++ !setting), *s = 2))
         x := set(aa)
         every pair[!x] +:= 1
         }
      write(repl("-",12))
      }
   analyze(pair, labels)
   }
end
```

```
# Shuffle the players.

procedure shuffle(x)
   x := string(x)
   if not(type(x) == ("string" | "list")) then xstop(x)
   every !x :=: ?x
   return x
end

procedure xstop(x)
   stop("Run-time error 102 in shuffle: ", image(x))
end

# Add everyone else.

procedure singles(s, c)
   S := set([])
   every insert(S, c ~== !s)
   return S
end

# Make a selection.

procedure select(s1, base, setting)
   local s2
   if s2 := member(zero[s1], !base) then {
      every delete(zero[!setting], s2)
      every delete(zero[s2], !setting)
      }
   else if s2 := member(one[s1], !base) then {
      every delete(one[!setting], s2)
      every delete(one[s2], !setting)
      }
   else fail
   return s2
end

procedure remove(s1, s2)
   (s1[find(s2, s1)] := "") | stop("cannot remove")
   return s1
end

procedure displays(s)
   every writes(right(find(!s, letters),3))
   write()
end
```

```
# See how well the assignment worked.

procedure analyze(t, s)
   local hits, notes
   hits := list(10,0)
   notes := list(10,"")
   write("number of different pairings is ",*t)
   every pair := string(!s ++ !s) & *pair = 2  do {
      score := t[pair]
      hits[score + 1] +:= 1
      t[pair] := t[reverse(pair)] := 10
      if (score = 0) | (score > 2) then
         notes[score + 1] ||:= pair
      }
   write("pairings:")
   every i := 1 to 10 do
      write(i - 1,":","\t",hits[i])
   write("\n","notes:")
      every i := 0 | (3 to 10) do
         write(i,":","\n",xlate(notes[i + 1]))
end

procedure xlate(s)
   if *s = 0 then fail
   s1 := ""
   every i := 1 to *s by 2 do
      s1 ||:= right(find(s[i],letters),3) || right(find(s[i + 1],letters),3) ||
         "\n"
   return s1
end
```

```
#
#       E I G H T  -  Q U E E N S (2)
#
#
#  This program solves the eight-queens problem iteratively,
#  and makes no use of goal-directed evaluation.
#

global up, down, rows, x

procedure main()
   local i

   up := list(15)
   down := list(15)
   rows := list(8)
   x := list(8)

   i := 1
   while i <= 15 do {
      up[i] := down[i] := 0
      i +:= 1
      }
   i := 1
   while i <= 8 do {
      rows[i] := 0
      i +:= 1
      }
   queens2()
end
```

```
procedure queens2()
   local r,c

   c := 1
   r := 0
   while 1 do {
       while r <= 8 do {
          r +:= 1
          if (rows[r] = up[r - c + 8] = down[r + c - 1] = 0) then {
             x[c] := r
             if (c = 8) then print()          # print
             else {                           # advance c
                rows[r] := up[r - c + 8] := down[r + c - 1] := 1
                c +:= 1
                r := 0
                }
             }
          }
       if c = 1 then return                   # done
       else {                                 # retreat c
            c -:= 1
            r := x[c]
            rows[r] := up[r-c + 8] := down[r + c - 1] := 0
            }
    }
end

procedure print()
  local k

     k := 1
     while k <= 8 do  {
        writes(x[k]," ")
        k +:= 1
        }
     write()
end
```

REFERENCES

1. H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.
2. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.
3. A. Barr and E. A. Feigenbaum (ed.), in *The Handbook of Artificial Intelligence*, Vol. 2, HeurisTech Press , Stanford, California, 1982.
4. M. Bellia and G. Levi, "The Relation Between Logic and Functional Languages: A Survey", *The Journal of Logic Programming Vol.3*(1986), 217-236.
5. M. Broynooghe and L. M. Pereira, Deduction revision by intelligent backtracking, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Ltd., Chichester, 1984.
6. T. A. Budd, "An Implementation of Generators in C", *J. Computer Lang. 7*(1982), 69-87.
7. J. Campbell (ed), *Implementations of Prolog*, Ellis Horwood, 1984.
8. J. Cohen, "Describing Prolog by its Interpretation and Compilation", *Comm. ACM 28*, 12 (Dec. 1985), 1311-1324.
9. S. K. Debray and D. S. Warren, Detection and Optimization of Functional Computations in Prolog, in *Proceedings of the 1986 Symposium on Logic Programming*, 1986.
10. J. N. Doyle, *A Generalized Facility for the Analysis and Synthesis of Strings and a Procedure-Based Model of an Implementation*, SNOBOL4 Project Document S4D38, University of Arizona, Feb. 1975.
11. A. P. Ershov, "On the essence of compilation", in *Formal Description of Programming Concepts*, E. J. Neuhold (ed.), 1978, 391-420.
12. E. Gallesio, *Inclusion de L'Evaluation Dirigee 'par Le But Dans Un Langage de Programmation Monomorphique*, Doctoral Dissertation, University of Nice, 1986.
13. S. L. Graham, P. B. Kessler and M. K. McKusick, "An Execution Profiler for Modular Programs", *Software—Practice & Experience Vol. 13*(1983), .
14. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1971.
15. R. E. Griswold and D. R. Hanson, "An Alternative to the Use of Patterns in String Processing", *ACM Trans. Prog. Lang. and Systems 2*, 2 (1980), 153-172.
16. R. E. Griswold, D. R. Hanson and J. T. Korb, "Generators in Icon", *ACM Trans. Prog. Lang. and Systems 3*, 2 (Apr. 1981), 144-161.
17. R. E. Griswold, "The Evaluation of Expressions in Icon", *ACM Trans. Prog. Lang. and Systems 4*, 4 (Oct. 1982), 563-584.
18. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
19. R. E. Griswold, "The Control of Searching and Backtracking in String Pattern Matching", in *Implementations of Prolog*, J. Campbell (ed.), Ellis Horwood, 1984, 50-64.
20. R. E. Griswold and W. H. Mitchell, *A Tour Through the C Implementation of Icon; Version 5.10*, The Univ. of Arizona Tech. Rep. 85-19, 1985.

21. R. E. Griswold, *The Icon Program Library; Version 6, Release 1*, The Univ. of Arizona Tech. Rep. 86-13b, 1986.

22. R. E. Griswold and M. T. Griswold, *The Implementation of The Icon Programming Language*, Princeton University Press, 1986.

23. D. Gudeman, *A Continuation Semantics for Icon Expressions*, The Univ. of Arizona Tech. Rep. 86-15, 1986.

24. C. T. Haynes, D. P. Friedman and M. Wand, "Obtaining Coroutines with Continuations", *J. Computer Lang. Vol. 11*, 3/4 (1986), 143-153.

25. C. Hewitt, PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot, in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1971.

26. N. D. Jones, P. Sestoft and H. Sondergaard, "An Experiment in Partial Evaluation: The Generation of a Compiler Generator", *Lecture Notes in Computer Science Vol. 202*(May 1985), 124-140.

27. P. Klint, An Overview of the SUMMER Programming Language, in *Conference Record of the Seventh Annual ACM Symp. on Prin. of Programming Languages*, 1980.

28. J. T. Korb, *The Design and Implementation of A Goal-Directed Programming Language*, Doctoral Dissertation, The University of Arizona, 1979.

29. R. Kowalski, "Predicate Logic as a Programming Language", in *Proceedings of the International Federation of Information Processing Congress, 74*, 1974.

30. D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams, ORBIT: An Optimizing Compiler for Scheme, in *Proceedings of the SIGPLAN Notices 1986 Symposium on Compiler Construction*, 1986, 219-233.

31. B. Liskov, *CLU Reference Manual*, Springer-Verlag, 1981.

32. J. McCarthy, P. W. Abrahams, D. J. Edwards and M. I. Levin, in *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, MA, 1962.

33. C. S. Mellish, "Some Global Optimizations for a Prolog compiler", *The Journal of Logic Programming Vol. 2, No. 1*(April, 1985), 43-66.

34. A. Newell, *Information Processing Language-V Manual*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1961.

35. F. G. Pagan, "Converting Interpreters into Compilers", *Software—Practice & Experience Vol. 18(6)*(June 1988), 509-527.

36. T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.

37. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, *Programming with Sets: an Introduction to SETL*, Springer Verlag, 1986.

38. G. Smolka, "Fresh: A Higher-Order Language with Unification and Multiple Results", in *LOGIC PROGRAMMING: Functions, Relations, and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986, 469-524.

39. G. L. Steele, *Rabbit: A Compiler for Scheme*, AI Memo 474, MIT, 1978.

40. L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

41. J. Stojanovksi, "A Note on Implementing Prolog in Lisp", *Inf. Proc. Letters 23*(Nov. 1986), 261-264.

42. G. J. Sussman and D. V. McDermott, "From PLANNER to CONNIVER—A Genetic Approach", in *Proceedings of Joint Computer Conference 41, Part II, Vol. 41*, 1972, 1171-1179.

43. W. M. Waite and G. Goos, *Compiler Construction*, Springer-Verlag, New York, 1984.

44. K. Walker, *A Type Inference System for Icon*, The Univ. of Arizona Tech. Rep. 88-25, 1988.

45. S. B. Wampler, *Control Mechanisms for Generators in Icon*, Doctoral Dissertation, The University of Arizona, 1981.

46. S. B. Wampler and R. E. Griswold, "The Implementation of Generators and Goal-Directed Evaluation in Icon", *Software—Practice & Experience 13*, 6 (June 1983), 495-518.

47. D. H. D. Warren, *Implementing Prolog—Compiling Predicate Logic Programs*, Research Report 39, University of Edinburgh, 1977.