A Recursive Interpreter for Icon*

Janalee O'Bagy

TR 87-2

Janauary 19, 1987; Corrected March 2, 1987

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

A Recursive Interpreter for Icon

1. Introduction

Icon is a programming language designed for string and list processing and general high-level programming tasks. Icon generalizes the traditional view of expression evaluation in which expressions evaluate to a single result. In Icon, expressions may produce a sequence of results. 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 implementation of generators and goal-directed evaluation requires automatic control backtracking. However, because expression evaluation is limited lexically, a full coroutine facility is not required*. Several implementations of Icon have been developed. All of them are based on a similar virtual machine with a stack-based architecture; an interpreter executes the virtual machine instructions. However, in these implementations the core of the expression evaluation mechanism is intricate and complicated and obscures the simple properties of the underlying semantics. This report describes a new approach to implementing expression evaluation based on recursive interpretation. This interpreter provides a basis for implementing goal-directed evaluation that is straightforward and conceptually clear. Furthermore, the recursive implementation does not sacrifice speed and is comparable to previous implementations in efficiency.

After a brief discussion of implementation issues, previous models for implementing the expression evaluation mechanism are discussed to give perspective on the development of ideas in this area. Following this, the recursive interpreter method is presented. An understanding and programming familiarity with Icon is assumed. In particular, the reader should be familiar with the terms *generator*, *goal-directed evaluation*, *control backtracking*, and *result sequence* and have a clear understanding of their meanings in Icon. Descriptions of these terms and of Icon expression evaluation semantics are given in [1,2].

2. Comments on Icon Expression Evaluation

The goal-directed evaluation process is founded on the concept that an expression may have an alternative result. In a language in which expressions produce exactly one result, or have only only way of being satisfied, goal-directed evaluation is meaningless. If an expression fails, there is no other way for evaluation to proceed except beyond the expression.

Success/Failure. In Icon, an expression may fail to produce a result, it may produce exactly one result, or it may produce many results. If an expression produces a result (or more than one) it *succeeds*, otherwise the expression *fails*. Success or failure of expressions drives the control structures of the language. An example is,

```
if find(s1,s2) = i then write(i)
```

Any expression can be used in control clauses of control structures.

Goal-Directed Evaluation. The embedded control structure for the evaluation of all expressions is *goal-directed evaluation*. The evaluation mechanism attempts to produce a result for all expressions. If failure occurs during the evaluation of the expression, the evaluation mechanism resumes previous components of an expression. The resumption of expressions on failure is automatic in the goal-directed evaluation process. Consider, for example, the implied control flow of this Icon expression:

```
s == !a
```

Suppose that s is a string and that a is a list of strings. Then s is compared to each element of a until an identical string is found. In a language such as C, by contrast, the implied control flow and state information would be coded

*The co-expression facility is not discussed in this report.

explicitly by the programmer. The variable telling the current index into the array would be named and the control would be detailed by a for or while loop.

Order of Evaluation. 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 and gives all possible results of combining results left-to-right with right-most results produced first. 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}.

Bounded Expressions. An important semantic feature of Icon concerns the scope of its goal-directed evaluation mechanism. Rather than having the entire program, or even a procedure body, consist of one monolithic expression, Icon programs consist of separate *bounded* expressions. Control backtracking is limited to the scope of a bounded expression. Once the bounded expression produces a result, it cannot be resumed for another. For example, suppose the following two lines occur in a procedure body:

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

Once the first line has produced a result, it cannot be resumed for another. Thus the outcome of evaluating p() cannot affect the evaluation of the previous line. If p() fails, the generator find is not resumed.

Bounded expressions are the basic structural components of goal-directed evaluation. 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. The implicit bounded expressions of Icon are:

- Each expression of a procedure body is bounded.
- In a compound expression {$expr_1$; $expr_2$; ...; $expr_n$ }, all expressions except $expr_n$ are bounded.
- The control expression of an if expression is bounded.
- The control and do expressions looping control structures are bounded. For example, in while $expr_1$ do $expr_2$, both $expr_1$ and $expr_2$ are bounded.

Note that since the expressions of a procedure body are bounded, every procedure consists of at least one bounded expression. Furthermore, during evaluation, bounded expressions become nested, as in

```
while expr1 do {
   if expr2 then expr3
   expr4
   }
```

Generators. Generators bring two issues to the implementation. First, the internal state information of the generator must be maintained. For example, in s == !a, the element generator ! must keep track of the position in a in order to successively generate its elements. The local state of the generator must be preserved between suspension and resumption.

The second issue is subtler: generators prolong the lifetime of temporary values. For example, in s == !a, wherever the operands of the comparison operation reside, those locations cannot be released; the possible resumption of the element generation operator ! causes the reevaluation of the string comparison operation. The first element of a may not be equivalent to s. Since the comparison is performed again with subsequent elements of a, the left operand must still reside in the same location. The solution to the prolonged lifetime of temporaries ultimately depends on the design of the virtual machine (or the evaluator) for Icon. If the Icon virtual machine is a stack machine where operations get their operands from the stack, those operands must be replicated when generators are present, since operations consume their operands. If the Icon machine is a temporary register machine, the values of registers must be preserved throughout the scope of the expression.

A further point about generators is that a generator implicitly provides a control backtracking point for an expression. 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 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.

The implementation issues can be summarized as follows:

- The backtracking points due to nested bounded expressions and generators must be maintained in stack order.
- A uniform mechanism to resume generators when failure occurs is needed.
- The state information for generators must be preserved.
- Generators extend the lifetime of temporary values.

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 structure 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 Icon expression evaluation.

3. The Icon Virtual Machine

All of the implementations of Icon discussed in this report are based on a similar virtual Icon machine. Icon source programs are translated into a virtual machine code, which is then interpreted. The virtual machine is based on a stack architecture; expressions are translated into post-fix notation. Thus operations get their corresponding arguments from the stack and replace the arguments with the result of the operation. An example of code for the simple addition expression i + 5 is:

```
var     i
int     5
plus
```

The code for a generative operator or function is similar. There is no distinction in the generated code for generators. Thus the code for 1 to 10 by 1 is:

```
int     1
int     10
int     1
toby
```

As explained previously, the inherent goal-directed control structure is founded on the concept of a bounded expression in which control backtracking takes place. In the virtual machine code, a bounded expression *$expr_1$* is represented by the code

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

If *$expr_1$* fails, execution continues at the code at label L1. If *$expr_1$* produces a result, the unmark is reached and execution continues at the instruction following the unmark. The 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 if expression

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

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. The selected expression is evaluated in whatever context the if expression occurs.

A looping control structure is used to repeatedly evaluate an expression as long as the control clause succeeds. Therefore, looping control structures ultimately fail. A different form of the mark instruction, which does not have an explicit failure label, is used for the control clause of a loop. The instruction is mark0, and it transmits failure to the surrounding context. For example,

```
while expr1 do expr2
```

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 code for generative control structures has two variations in the implementations discussed in this report. In two of the implementations, the virtual machine has an instruction, esusp, which corresponds to the concept that an expression is suspending. The generative control structures are then translated into code consisting of both mark and esusp instructions. For example, the alternation expression *expr*$_1$ | *expr*$_2$ is translated into

```
            mark        L1
            code for expr1
            esusp
            goto        L2
L1:
            code for expr2
L2:
```

In two other implementations, the code for alternation is:

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

The first form uses the instruction esusp to perform the actions needed to suspend a generator, while the second form uses a distinct instruction for each generative control structure and the suspension is done at that point. The recursive interpreter uses the latter form.

4. Previous Models of Implementation

Several implementations of Icon have been developed since the original version written by the designers of Icon. This section briefly discusses three significant versions that have evolved over the years. All use the stack-based architecture for the Icon virtual machine. They differ 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 Icon values. Also, the use of the system stack varies, and some of the implementations use an auxiliary stack for maintaining information.

Versions 1 and 2

The original implementation of Icon, written in Fortran, [3,4] uses two stacks. The interpreter stack holds active procedure frames and the temporary Icon values due to expression evaluation. A second stack, called the control stack, holds control and state information for goal-directed evaluation.

A global variable holds the failure label for the current bounded expression. 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; also the heights of the interpreter and control stacks are 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 recently stack failure label is restored from the stack.

Generators use the control stack to maintain local state between suspension and resumption. Generators are coded so that each *call* to the generator produces the next result. Thus they follow a coding 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 on 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 failure 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.

Notice that 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 coding of generators is highly specialized and differs from other routines throughout the run-time system.

Versions 3 through 5

Versions 3 through 5 of Icon [5-7], which are written in C and assembly language, combine the interpreter and control stacks on the system stack used by C. The 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 throughout interpretation.

Information for expression evaluation is maintained in two types of frames: expression frames and generator frames. Global pointers are maintained to point to the current expression and generator frames. Whenever a bounded expression is entered, a new expression frame is created on the system stack. 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 looks at the current values of the expression and generator frame pointers. If a generator is present, the interpreter restores the values from 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 complicated. For example, generative 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.

Version 6

Version 6 of Icon [8] is similar to Version 5, but it is written entirely in C. This transformation is accomplished by introducing recursion into 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), Version 6 has the generator call 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 interpenter 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 the system stack.

Comments

The fundamental problem with these implementations is that 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 in recursive calls to the interpreter. The explicit methods of previous implementations are not necessary.

5. The Recursive Interpreter Model

The recursive interpreter focuses on the notion of alternatives in the evaluation process. Since alternative computational paths are taken when failure occurs and the interpreter must know where to continue in the code on failure, alternatives introduce *failure continuations.* An interpretive process that recurses when generators are encountered implicitly preserves both the internal state of the generator and control information for lifo resumption. Bounded expressions, although they do not introduce alternatives, also provide failure continuations. Through generators and bounded expressions, all control flow for Icon expression evaluation is expressed in terms of failure continuations.

The recursive interpreter is simple and uniform. It 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 interperter 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 the levels of recursion built up during the evaluation of the bounded expression and implements limiting the expression to one result during goal-directed evaluation.

The Icon 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 interpreter stack 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 the evaluation context. The interpretive process follows the method outlined above.

5.1 The Interpreter

The interpreter described in the following sections is presented in Icon as a prototype of the actual implementation. Describing a prototype avoids unnecessary detail. To further simplify the explanation, the issues of Icon procedure invocation and local variables are omitted. These issues are relatively uninteresting and are implemented in the standard way by pushing Icon local variables on the stack and maintaining a pointer to the local environment. The model for expression evaluation is unaffected by local Icon state. In all the examples, variables are global variables.

The interpreter routine implements the virtual machine. It requires code to interpret, a stack to hold Icon values, and the ability to perform the primitive operations of the instructions. The global variables of the interpreter are described below:

- icode—a list holding the virtual machine instructions
- ipc—an index into icode
- stack—the interpreter (virtual machine) stack used to hold Icon values
- globals—a table consisting of the global variables of the Icon program

Each invocation of the interpreter is an evaluation context for an expression. Variables local to the interpreter are used to maintain the expression context:

```
procedure interp(ep,sp)
   local fipc
      .
      .
      .
end
```

The variable ep points to the portion of the interpreter stack where the values relevant to the current expression begin, and the sp points to the current top of the interpreter stack. The variable fipc is the failure continuation for the current expression; it is simply an index into the icode list of virtual instructions.

Invocations of the interpreter accumulate as new expression contexts are encountered. The interpreter returns under two situations: when failure occurs or when the end of a bounded expression is reached. 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 and indicates that goal-directed evaluation in the most recent bounded expression is complete.

The structure of the Icon program implementing the prototype interpreter is:

```
global stack, icode, ipc
global iglobals

record var(v)

procedure main()
   init()
   interp(0,0)
end

procedure interp(ep,sp)
   local fipc, signal

   repeat {
      case FetchInst() of {
         "var"      :
             .
             .
             .
         "int"      :
             .
             .
             .
         "plus"     :
             .
             .
             .
         "mark"     :
             .
             .
             .
         }
end
```

The interpreter consists of a simple loop that fetches instructions and selects the appropriate code. The FetchInst procedure increments the ipc and returns the instruction pointed to by its original value. A corresponding FetchOpnd procedure returns the operand of an instruction.

Simple instructions, such as an integer or string literal, push and pop values from the interpreter stack. For example, the code in the interpreter for int is:

```
"int"  : {
   sp +:= 1
   stack[sp] := FetchOpnd()
   }
```

When an Icon source variable is referenced, the interpreter pushes a record on the stack whose value is the name of the variable. The code for var is:

```
"var" : {
   sp +:= 1
   stack[sp] := var(FetchOpnd())
   }
```

Variables are dereferenced when required by context. For example, operations dereference their operands. The procedure DeRef dereferences a variable by looking up the corresponding value in the globals table.

Operations use the values from the stack as their operands. Operations in Icon are either monogenetic, conditional, or generative. Monogenetic operations produce exactly one result. For example, the arithmetic operations +, *, etc., are monogenetic. The code for the plus instruction typifies these operations:

```
"plus" : {
   DeRef(sp -1)
   DeRef(sp)
   stack[sp-1] := stack[sp-1] + stack[sp]
   sp -:= 1
   }
```

Notice that the arguments are replaced by the result of the addition and that the sp is decremented. Execution continues at the beginning of the interpreter loop.

Conditional operations, on the other hand, may produce a result or they may fail to produce a result. In the case that a result is produced, interpretation is similar to a monogenetic operation, like addition above. However if the operation fails, interp returns a Resume signal to continue goal-directed evaluation in a previous context. For example, the code for the numerical comparison operation < is

```
"numlt" : {
   DeRef(sp -1)
   DeRef(sp)
   if stack[sp-1] >= stack[sp]
      then return Resume
   stack[sp-1] := stack[sp]
   sp -:= 1
   }
```

The remaining generative operations are discussed in later sections.

When control decisions are encountered, either directly in the icode or in the primitive operations, the current context of the expression environment is saved and the interpreter is invoked to continue execution in a new environment. There are several situations that require a new context to be created: bounded expressions, generative control structures, and generative operations and functions. Interpreting a bounded expressions gives the first example of the basic control pattern of the interpreter.

5.2 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 may be removed. This information is of two kinds: the values on the interpreter stack that accumulate during evaluation, and the recursive invocations of the interpreter due to generators in the expression. Both kinds of information must be removed when the expression is complete.

The ep points to the base of the stack for the expression currently being evaluated. Since bounded expressions occur only at control decision points, a bounded expression does not need to be connected with the values that may currently reside on the interpreter stack. Thus when evaluation begins for a bounded expression, the ep is adjusted to point to a "fresh" portion of the stack.

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

```
                mark          L1
                code for expr
                unmark
                    .
                    .
                    .
        L1:
```

The mark instruction "marks" the boundary for control backtracking and the label L1 is the failure continuation, the place in the icode to continue execution if the expression *expr* fails. If the expression succeeds, the unmark instruction is reached and the interpreter removes the context of *expr*. The code for mark follows:

```
"mark" : {
   fipc := FetchOpnd()                    # get the failure continuation
   if interp(sp+1,sp) = Resume then       # expression failed
      ipc := fipc
   }
```

At the mark instruction, the failure continuation L1 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 stack. The instructions of the bounded expression are interpreted in the new context.

To illustrate this, consider evaluating an if expression whose result is the larger of two strings:

```
if s1 << s2 then s2 else s1
```

The corresponding virtual machine code is:

```
            mark          L1
            var           s1
            var           s2
            lexlt
            unmark
            var           s2
            goto          L2
      L1:
            var           s1
      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 s1 and s2 are pushed onto the stack and the lexical comparison operation is performed. For the moment, suppose that s1 *is* lexically less than s2. Then the comparison succeeds and lexlt decrements the sp, leaving s2 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 s1 << s2 leaves only one result result on the stack and does not incur any new invocations of the interpreter.

The code for unmark is simply

```
"unmark" : return Clear
```

After the return, control returns to the interpreter at the mark instruction. 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. Notice that since ep and sp are local variables in the interpreter, they are automatically reset to their previous values by the return. Thus the contents of the stack are as they were before the bounded expression was evaluated. As execution continues, s2 is pushed on the stack and it becomes the result of the if expression.

5.3 Failure

In the example above, the expression produces a result and unmark signals that the context of the expression is to be removed by returning the appropriate signal. Likewise, 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. If there are no generators, the most recent control point is the failure continuation of the current bounded expression.

Contrast the execution in the previous section with one in which failure occurs. Using the same example, suppose now that s1 is lexically greater than s2 so that s1 << s2 fails. The lexical comparison operation is conditional and is coded like all the conditional operations:

```
"lexlt" : {
   DeRef(sp -1)
   DeRef(sp)
   if stack[sp-1] >>= stack[sp]
      then return Resume
   stack[sp-1] := stack[sp]
   sp -:= 1
   }
```

Execution proceeds as in the previous example upto the lexlt instruction. Since the comparison fails, lexlt returns the signal Resume. Control returns to the code in the interpreter at the mark instruction. 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 s1, 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 label 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 the goal-directed evaluation. Resumption of generators is straightforward and follows naturally from the method of recursing at failure continuation points.

5.4 Generators

A generator provides an alternative computational path during goal-directed evaluation. Specifically, it introduces a failure continuation. Recursion is the basic mechanism used to encode failure continuations for control backtracking. The recursive interpreter makes no distinction between the failure continuations of bounded expressions and those of generators, and handles them similarly.

However, besides recursing to "stack" its failure continuation, a generator must also address the lifetime problem of temporary Icon values (see Section 2). In a given context of evaluation, the ep points to the base of the values on the interpreter stack that are relevant to the current expression context. Therefore, just before a generator invokes the interpreter, it copies the values on the interpreter stack from the current ep to the sp. In the new context, the 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 interpreter 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.

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.

All varieties of generators in Icon—operators, built-in 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 interpreter stack, and calls the interpreter recursively to stack its failure continuation and establish a new context.

The alternation control structure illustrates the combination of copying, recursion, and signals used in the implementation. The generated code for the expression

expr$_1$ | *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 *expr2*. The code for alternation is:

```
"alt" : {
   fipc := FetchInst()                # get the failure continuation
   newsp := copy(ep,sp)               # copy the values from the ep to sp
   signal := interp(sp+1,newsp)
   if signal = Resume then            # evaluate alternative expression
      ipc := fipc
   else
      return signal
   }
```

The interpreter uses the procedure copy to replicate the values on the interpreter stack. This procedure returns the new value of the stack pointer after the copy.

Consider evaluating the expression s1 <<= (s2 | s3), which succeeds if either s2 or s3 is lexically greater than s1. The code for the expression is:

```
        global      s1
        alt         L2
        global      s2
        goto        L3
L2:
        global      s3
L3:
        lexle
```

After the first global instruction, 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 s1, depending on the context of the expression s1 <<= (s2 | s3) in the source code. Execution continues in a new context at the global s2 instruction. The goto avoids evaluating the alternative expression.

Now the lexle is performed. There are two important points to notice. First, lexle operates on replicated values. Any computations using values on the stack in this 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 expression. Thus if the comparison fails, control returns to alternation and the second alternative is attempted.

The uniformity of the interpretive process makes goal-directed evaluation straightforward. In the example above, if the first attempt of the lexical comparison succeeds, the result of the expression s1 <<= (s2 | s3) becomes the value of s2. 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.

5.5 Generative Operators and Functions

The distinction between operators and functions is syntactic only. This section describes the implementation of the generator *expr1* to *expr2*. Generative functions such as find and upto are implemented in the same way as the to generator.

The code generated for *expr₁* to *expr₂* is

```
code for expr1
code for expr2
int          1
toby
```

Note that the default increment value is 1; the translator supplies the increment value if it is omitted. Before the toby instruction is reached, its operands are evaluated and their results are left on the interpreter stack.

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

The code for toby follows:

```
"toby" : {
   fipc := ipc                           # set the failure continuation
   DeRef(sp-2)
   DeRef(sp-1)
   DeRef(sp)
   from := stack[sp - 2]
   limit := stack[sp -1]
   inc := stack[sp]
   while from <= limit do {
      newsp := copy(ep,sp-3)             # copy current context
      stack[newsp+1] := from             # push result of toby
      signal := interp(sp + 1,newsp + 1)
      if signal ~= Resume then return signal
      ipc := fipc
      from := from + inc
      }
   return Resume
}
```

5.6 Generative Control Structures

The two remaining generative control structures are limitation and repeated alternation. These two control structures require an extension of the techniques seen so far. Briefly, the problem is that these control structures require knowing information that is hidden in the levels of recursion; communicating by signals is insufficient. Fortunately, the method used to implement these control structures is simple and does not burden the interpreter with additional mechanisms and state information.

Repeated Alternation

The result sequence for |*expr₁* is the repeated concatenation of the result sequence for *expr₁*. For example, the result sequence for |"a" is

{ "a", "a", "a", ... }

and the result sequence for |(1 to 3) is

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

The result sequence for *expr₁* is produced repeatedly *unless* at some point the evaluation of *expr₁* fails. Only expressions whose outcomes depend on side effects or global variables may succeed on one evaluation and fail on another. For example, evaluating read() succeeds for each line in the input, and fails when the end of the input is reached. Thus |read() repeatedly reads until end-of-file causes evaluation of read() to fail. In evaluating |$expr_1$, if repeated alternation did not check for failure of $expr_1$, execution could never continue past the expression. 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.

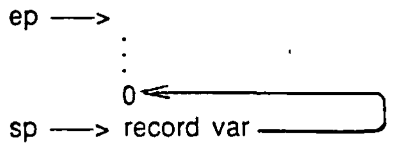
The difficulty in implementing repeated alternation is that the two instructions repalt and contrep must communicate. So far, the method used for communicating has been returning signals between contexts of interpretation. However, contrep cannot return to the context at repalt, since returning removes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

A possible solution is to use a global variable for communicating. 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 recursive interpreter uses a simpler method to achieve communication. It constructs a variable on the interpreter stack when an instance of repeated alternation is evaluated. So far in the discussion of the implementation, only Icon global variables have been used. For an Icon source variable, the interpreter pushs a record on the stack whose value is a string naming an Icon variable. To construct a local variable, the interpreter pushes a record whose value is an index into the interpreter stack. The code sequence below constructs a such a variable:

```
sp +:= 1
stack[sp] := 0
sp +:= 1
stack[sp] := var(sp-1)
```

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

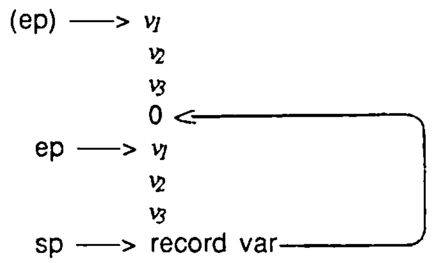


Whenever an instance of repeated alternation is evaluated, the interpreter constructs a variable on the interpreter stack that is associated with that instance of the control structure. The corresponding contrep instruction need only know where that variable is when it gains control. The uniformity of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

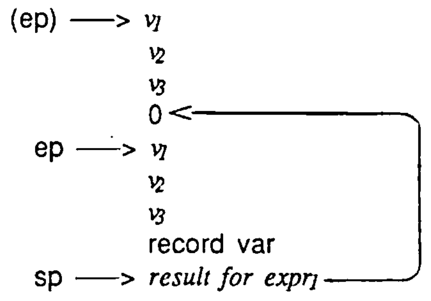
To illustrate, suppose that the interpreter stack has the following form when repeated alternation is encountered:

```
ep ——> v1
       v2
sp ——> v3
```

Since repeated alternation is a generator, it follows the general protocol: it saves the failure continuation for the expression, it copies from the current ep to sp, and it 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 which 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 value from the top of the stack. On other hand, the expression *expr₁* 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 *expr₁*, the top of the stack still has the form above; the repeated alternation variable is the second word from the top of the stack.

The code for repeated alternation is:

```
"repalt": {
   fipc :=  ipc                       # establish the failure continuation
   sp +:= 1                           # make room for the control structure value
   repeat  {
      ipc := fipc
      stack[sp] := 0                  # set the repalt value to 0
      newsp := copy(ep,sp - 1)
      stack[newsp+1] := var(sp)       # construct a variable pointing to repalt value
      signal := interp(sp+1,newsp+1)
      if signal ~== Resume then return signal
      if stack[sp] = 0 then return Resume   # contrep was not reached
      }
```

The code at repalt 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 succeeded. The code for contrep follows:

```
"contrep" : {
   stack[stack[sp-1].v] := 1          # change repalt value to 1
   stack[sp-1] := stack[sp]           # remove variable
   sp -:= 1
   }
```

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. Otherwise, this variable would interfere with subsequent computations.

Limitation

Unlike alternation, limitation is not a generator; instead it limits 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. Limitation can be thought of as a generalization of a bounded expression. The generated code for $expr_1 \setminus expr_2$ is

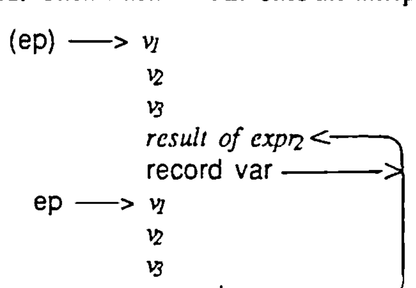
```
code for expr2
limit
code for expr1
lsusp
```

In order to control the generation of results for $expr_1$, limit and lsusp must cooperate. The lsusp instruction must signal to remove the expression context only when the expression produces its last permitted value. The same mechanism for communicating used in repeated alternation is used for the limitation control structure. limit pushes a variable on the interpreter 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 actions depend 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. The code in the interpreter for limitation follows:

```
"limit" : {
   DeRef(sp)
   if stack[sp] = 0 then return Resume      # expression limited to 0 results
   sp +:= 1
   stack[sp] := var(sp-1)                   # push a variable that points to limit counter
   newsp := copy(ep,sp-2)
   signal := interp(sp+1,newsp)
   if signal = Limit & stack[sp] = 0 then
      sp -:= 1
   else
      return signal
   }
```

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 value with this result and also replaces the first variable with the value 0 to indicate that this instance of limitation is complete. (Limitation expressions may be nested; the Limit signal must propagate to the appropriate occurrence of limit.) The code for lsusp follows:

```
"lsusp":  {
   i := stack[sp-1].v                  # get pointer to limit value
   stack[i] -:= 1                      # decrement limit value
   if stack[i] ~= 0 then {             # last result not yet produced
      stack[sp-1] := stack[sp]
      sp -:= 1
      }
   else {                              # last allowed result has been produced
      stack[i] := stack[sp]            # replace limit counter with result
      stack[i+1] := 0                  # replace variable
      return Limit
      }
   }
```

By using the variable pointing to the limit value (now the second word 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 *$expr_1$*.

Iteration

In every *$expr_1$*, the expression *$expr_1$* is repeatedly resumed until its result sequence is complete. 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
```

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:

```
"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 code for every *$expr_1$* do *$expr_2$* is:

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

6. Performance of the Recursive Interpreter

Since the interpreter described here calls itself recursively whenever there is a new context for expression evaluation, such as at the beginning of a bounded expression and when a generator suspends, it is important to know what effect this recursion has on the performance of interpretation and the resources it requires.

There are two main issues: the cost of recursive calls in terms of time and the amount of stack space needed. It is not possible to compare the recursive interpreter with all of its successors in a meaningful way, since their performance depends on many matters that are not related to the issues here. However, some valid comparisons can be made.

The Version 6 interpreter, which is in use at present in the publically distributed version of Icon, is written entirely in C and uses recursion to some extent, as described in Section 4. It is about 5% to 10% slower than its predecessor, which relied on assembly language code to perform manipulations on the system stack that did not conform to ordinary stack protocols. In addition, the interpreter loop itself in the predecessor was written in assembly language. The difference in performance between these interpreters probably is attributable to the use of assembly language rather than C.

The recursive interpreter also is written entirely in C and is structurally similar to the Version 6 interpreter, except for the more general use of recursion in place of the explicit construction of frames on the interpreter stack. The question, then, is the comparative cost of the two approaches to handling information that must be saved when generators suspend.

Timing tests on a VAX 8600 and a Sun-3 Workstation show no measurable difference in running speed between the two interpreters on a wide range of programs, although it is possible to contrive programs that favor one or the other interpreter.

There is, however, a noticeable difference in stack utilization. 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 higher for the recursive interpreter than for the Version 6 interpreter. On the other hand, the average high-water mark on the interpreter stack for the recursive interpreter is about one-half that of the Version 6 interpreter. It is worth noting that the interpreter stack is a C array and contributes significantly to the space needed to run user programs.

For computers with a limited 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 suspended at any one time. In Icon, this number typically is relatively small; a maximum of five is typical.

7. Conclusions

As indicated earlier, the recursive interpreter is completely implemented and operational. It currently is being used in an experimental version of Icon that is being used to develop new expression evaluation mechanisms.

The idea of using recursion in an interpreter is hardly new [9]. 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. 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 in evaluation and recursion in their implementation.

Performance is not a significant issue for the recursive interpreter. Its advantages apply primarily to possible extensions to Icon and the implementation of similar types of expression evaluation in other programming languages such as C [10] and Pascal [11]. While details vary for different languages and implementation frameworks, the same principles apply. In fact, the approach described here is not limited to interpretation. It can be applied to compilation as well, in which the code that is generated makes a similar use of recursion. The potential advantage of compilation is that optimizations can be performed that are not practical for an interpreter.

Acknowledgements

Ralph Griswold contributed substantially to the conceptual ideas in this model for implementing Icon expression evaluation, as well as to the presentation of the ideas in this report.

The first implementation of Icon was done primarily by Dave Hanson and Tim Korb. Cary Coutant and Steve Wampler implemented the interpreter that used assembly language to manipulate the system stack in non-standard ways. Bill Mitchell introduced recursion into the implementation of generators in Version 6.

Each of these implementations contributed to subsequent ones. The work described here owes a substantial debt to all of its predecessors.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. D. Gudeman, *A Continuation Semantics for Icon Expressions*, The Univ. of Arizona Tech. Rep. 86-15, 1986.
3. J. T. Korb, *The Design and Implementation of A Goal-Directed Programming Language*, Doctoral Dissertation, The University of Arizona, 1979.
4. 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.
5. S. B. Wampler, *Control Mechanisms for Generators in Icon*, Doctoral Dissertation, The University of Arizona, 1981.
6. 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.
7. 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.
8. R. E. Griswold and M. T. Griswold, *The Implementation of The Icon Programming Language*, Princeton University Press, 1986.
9. H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, 1985.
10. T. A. Budd, "An Implementation of Generators in C", *J. Computer Lang.* 7(1982), 69-87.
11. E. Gallesio, *The Programming Language* π, Draft Report, University of Nice, 1985.