Programmer-Defined Evaluation Regimes*

Michael Novak and Ralph E. Griswold

TR 82-16a

December 16, 1982; revised December 22, 1983

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

Programmer-Defined Evaluation Regimes

1. Introduction

The term *control operation* is used here to refer to any mechanism that affects the sequencing of expression evaluation [1]. For example, argument evaluation is typically considered to be a control operation. Most programming languages have a fixed set of control operations, and very few languages allow programmers to define new operations. The main reason is that while many new control operations can be constructed, they are generally combinations or modifications of a very small set of basic control operations.

In Icon, however, the potential of programmer-defined control operations is greater, since an expression can produce a sequence of results [2-4]. For example, in most conventional programming languages procedures are called in a manner similar to that of Icon. However, each set of values passed to a procedure in such programming languages produces a single result. Therefore in these languages, a procedure call activates the procedure a single time with an n-tuple consisting of the results produced by evaluating each expression in the argument list, where n is the size of the argument list. Only one n-tuple, and therefore only one procedure invocation, is possible. Consequently alternate evaluation regimes are not meaningful. In Icon, however, many possible sequences of n-tuples are possible. By constructing different evaluation regimes, different result sequences can be produced. The need for this added control over expression evaluation and hence result sequence production motivated a mechanism, called PDCO, for defining control operations [5].

The material that follows assumes a knowledge of Version 5 of Icon [2].

2. The PDCO Facility

The PDCO facility uses co-expressions to provide control over expression resumption [6]. A control operation is written in the form of a *control procedure*. By convention a control procedure has a single parameter that is a list of co-expressions. This allows a control procedure to act as if it has an arbitrary number of parameters (the elements of the list).

A control procedure p is called as

p{$expr_1$, $expr_2$, ..., $expr_n$}

This syntactic extension to Icon is equivalent to

p([create $expr_1$, create $expr_2$, ..., create $expr_n$])

Thus, the PDCO facility provides a convenient way to use co-expressions when invoking a control procedure.

The next sections review the use of the PDCO facility for modeling existing control operations and for creating ones. A more complete discussion is given in Reference 5.

2.1 Modeling Existing Control Structures

An example of the use of this facility is illustrated by alternation with

Alt{$expr_1$, $expr_2$}

which models

$expr_1$ | $expr_2$

Alternation is implemented by the control procedure

```
procedure Alt(a)
   local x
   while x := @a[1] do suspend x
   while x := @a[2] do suspend x
end
```

Alternation can be generalized to an arbitrary number of arguments:

```
procedure Galt(a)
   local e, x
   every e := !a do
      while x := @e do suspend x
end
```

so that

$expr_1 \mid expr_2 \mid \ldots \mid expr_n$

is modeled by

Galt{$expr_1, expr_2, \ldots, expr_n$}

2.2 New Control Operations

An example of a useful new control operation is Select. The operation

Select{$expr_1, expr_2$}

is similar to

$expr_1 \setminus expr_2$

except that in Select, $expr_2$ produces a sequence of positive integers that are in monotone nondecreasing order. For each i produced by $expr_2$, the ith result of $expr_1$ is produced. For example, the result sequence for

```
Select{11 to 15, 2 | 4 | 6}
```

is {12, 14}. Select is implemented by the control procedure

```
procedure Select(a)
   local  i, j, x
   j := 0
   while i := @a[2] do {
      while  j < i do
         if x := @a[1] then j +:= 1
         else fail
      if i = j then suspend x
      else stop("selection sequence error")
      }
end
```

3. Programmer-Defined Evaluation Regimes

Some simple programmer-defined evaluation regimes were introduced in Reference 5. These regimes are reviewed in the next section. Examples of their use and the introduction of more advanced features follow.

3.1 Procedure Invocation

In Icon, procedures are invoked by procedure calls of the form

p($expr_1$, $expr_2$, ..., $expr_n$)

where p is a procedure and $expr_1$, ..., $expr_n$ are expressions, each of which produces a sequence of results. Each procedure activation requires an n-tuple that consists of one result from the result sequence of each expression passed to the procedure. Note that this is very similar to procedure invocation in most conventional programming languages, except that expressions in most languages produce at most one result. For example, the Icon function

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

produces the positions in s2 where s1 occurs.

Similar functions exist or can be written easily in other programming languages. For example, the PL/1 function INDEX(s1,s2) returns the first position of s2 in s1. If s2 does not occur in s1, 0 is returned as a special value. For comparison with Icon, the following PL/1 procedure can be used:

```
find: PROCEDURE(s1, s2);
      RETURN(INDEX(s2, s1));
      END find;
```

A SNOBOL4 version of find(s1, s2) is

```
        DEFINE("find(s1, s2)place")
              .
              .
              .
find    s2   ARB @place s1     :F(FRETURN)
        find = place + 1       :(RETURN)
```

In the SNOBOL4 version, find fails if s1 does not occur in s2. Unlike Icon, neither PL/1 nor SNOBOL4 allows a function to produce more than one result. For example, the expression

```
find("a", "amalgamated")
```

produces the result 1 in both PL/1 and SNOBOL4, but has the result sequence {1, 3, 6, 8} in Icon.

To see the significance of the differences between these languages, consider the following Icon procedure that formats integer pairs.

```
procedure format(i, j)
   return "[" || i || ":" || j || "]"
end
```

For equivalent procedures in PL/1 and SNOBOL4

```
format(1, 3)
```

produces

```
[1:3]
```

in all cases, and

```
format(find("a", "amalgamated"), find("b", "babble"))
```

produces

```
[1:1]
```

in PL/1 and SNOBOL4. In Icon, however, this expression has the result sequence

```
[1:1]
[1:3]
[1:4]
[3:1]
[3:3]
[3:4]
[6:1]
[6:3]
[6:4]
[8:1]
[8:3]
[8:4]
```

This result sequence is dependent on the built-in Icon evaluation regime, which uses left-to-right evaluation with lifo resumption to produce the n-tuples with which a procedure is called.

Programmer-defined evaluation regimes make it possible to produce different result sequences by producing different sequences of n-tuples. Note that programmer-defined evaluation regimes would have little use in PL/1 or SNOBOL4, since expressions in these languages only produce one result.

To put this in perspective, consider the following programmer-defined evaluation regime that mimics the evaluation mechanism used by SNOBOL4.

```
procedure Simple(a)
   local i, x
   x := list(*a)                   # list for argument results
   every i := 1 to *a do
      x[i] := @a[i] | fail         # evaluate argument or fail
   return Call(x)                  # invoke procedure
end
```

Simple evaluates argument expressions from left to right and produces the first result produced by each argument expression. The procedure is then invoked with these results. Note that the procedure cannot be invoked directly, since the arguments to the evaluation regime actually are passed as a list of co-expressions, as described in Section 2. That is, given an argument evaluation regime R,

```
R{p, e1, , en}
```

is equivalent to

```
R([create(p), create(e1), , create(en)])
```

Therefore, Call is used to invoke the procedure:

```
procedure Call(a)
   suspend case *a of {
      1 : a[1]()
      2 : a[1](a[2])
      3 : a[1](a[2], a[3])
      4 : a[1](a[2], a[3], a[4])
      5 : a[1](a[2], a[3], a[4], a[5])
      6 : a[1](a[2], a[3], a[4], a[5], a[6])
                     :
      default :  stop("Call : too many args.")
      }
end
```

Call determines the number of arguments and invokes the procedure with these arguments. The use of Call is a byproduct of the way argument evaluation regimes are implemented in Icon. It has nothing to do with the operation of Simple, per se.

Note that the expression

```
Simple{format, find("a", "amalgamated"), find("b", "babble")}
```

has the result sequence

```
[1:1]
```

Although Simple uses only the first result produced by each argument expression, the result sequence produced by Simple in Icon may still be of size greater than one, since Icon procedures may produce many results for a single argument n-tuple. For example, the result sequence for

```
Simple{find, "a", "capable"}
```

is the result sequence for

```
find("a", "capable")
```

which is {2, 4}. On the other hand,

```
Simple{find | match, "a" | "b", "capable" | "believable"}
```

also has the result sequence {2, 4}, since only the first result of each argument to Simple is used.

The evaluation regime Simple does not resume argument expressions. However, an evaluation regime to do this is can be produced by a simple modification to Simple:

```
procedure Parallel(a)
   local i, x
   x := list(*a)                    # list for argument results
   repeat {
      every i := 1 to *a do
         x[i] := @a[i] | fail       # evaluate argument or fail
      suspend Call(x)               # invoke procedure
      }
end
```

In Parallel, every argument is resumed to produce a new n-tuple and evaluation terminates when any argument fails to produce a result. Note that arguments still are evaluated from left to right as in the built-in lifo regime. For generality, the first argument (the procedure) is evaluated in the same manner as all the other arguments. Therefore, since all argument expressions are resumed in "parallel", arguments such as format, that would be single values in Simple, must be generated repeatedly in a parallel resumption call. Thus, the expression

```
Parallel{|format, find("a", "amalgamated"), find("b", "babble")}
```

has the result sequence

```
[1:1]
[3:3]
[6:4]
```

Note that parallel resumption, unlike the built-in lifo resumption, does not produce all possible combinations of argument results.

The usefulness of parallel resumption is illustrated by the following call:

```
Parallel{|format, !&ucase, !&lcase}
```

The result sequence of the expression is a list of corresponding upper- and lowercase letters:

```
[A:a]
[B:b]
[C:c]
[D:d]
[E:e]
[F:f]
[G:g]
[H:h]
  :
  :
```

3.2 Writing Evaluation Regimes

An evaluation regime R is invoked by a call of the form

R{p, $expr_1$, ..., $expr_n$}

where p is the procedure to be invoked and $expr_1$, ..., $expr_n$ are expressions, each of which produces a sequence of results. The results produced are the arguments for p. Like any other Icon expression, both R and p can produce a sequence of results.

An evaluation regime is written as a control procedure consisting of two parts: (1) the evaluation of the arguments and (2) the invocation of the procedure.

The evaluation phase of a regime normally creates an n-tuple of results, one from each argument in the argument list. This is done by first evaluating each argument to produce a result. If an argument fails to produce a result, either another argument is resumed or the regime terminates. It is the particular evaluation regime being used that determines what argument, if any, should be resumed.

For example, the built-in lifo evaluation regime evaluates its arguments from left to right. If an argument does not produce a result, the previous argument is resumed. If there is no previous argument, the regime terminates. It is primarily lifo resumption that determines the order in which results are produced, although if evaluation were from right to left, side effects might change the result sequence. Note that the argument that produces the procedure is treated no differently from the other arguments.

Once an n-tuple of results is produced, the procedure is invoked. Assume the n-tuple

```
x1, ..., xn
```

has been produced. Then a procedure invocation of the form

```
x1(x2, ..., xn)
```

is performed using Call. Each result produced by the procedure is in turn produced by the regime. When the procedure x1 terminates, the argument evaluation phase of the regime is re-entered. Note that x1 may generate an infinite sequence of results, in which case the regime produces an infinite sequence of results.

3.3 Examples

3.3.1 Lifo Resumption

As described in Section 3.1, the evaluation regime that is built into Icon is left to right with lifo resumption. The following evaluation regime mimics this built-in regime.

```
procedure Lifo(a)
   local i, x, ptr
   x := list(*a)                      # list for argument results
   ptr := 1
   repeat {
      repeat
         if x[ptr] := @a[ptr]         # evaluate argument if possible
         then {
            ptr +:= 1
            (a[ptr] := ^a[ptr]) |     # refresh next argument
            break                     # or break if out of range
            }
         else if (ptr -:=  1) = 0     # set pointer to previous argument
              then fail               # or fail
      suspend Call(x)                 # invoke procedure
      ptr := *a                       # reset pointer
      }
end
```

Using this procedure, a call of the form

$expr_1(expr_2, expr_3, \ldots, expr_n)$

is modeled by

Lifo{$expr_1, expr_2, \ldots, expr_n$}

Note that the regime re-enters the argument evaluation phase if and only if Call terminates (that is, procedure invocation terminates). If the procedure never terminates, the regime never re-enters the argument evaluation phase and therefore it never terminates. This is the case in the built-in evaluation regime as well.

3.3.2 Reverse Evaluation

Consider an alternate argument evaluation regime that uses right-to-left evaluation:

```
procedure Reverse(a)
   local i, x, ptr
   x := list(*a)                      # list for argument results
   ptr := *a
   repeat {
      repeat
         if x[ptr] := @a[ptr]         # evaluate arg. if possible
         then {
            ptr -:= 1
            (a[ptr] := ^a[ptr]) |     # refresh next argument
            break                     # or break if out of range
            }
         else if (ptr +:= 1) > *a     # set pointer to previous arg.
              then fail               # or fail
      suspend Call(x)                 # invoke procedure
      ptr := 1                        # reset pointer
      }
end
```

If there are no side effects, the results in the result sequences for Reverse and Lifo are the same, although the order of the results may be quite different in the two cases. For example,

```
every Lifo{write, 1 to 5, !&lcase}
```

produces the following output:

```
1a
1b
1c
1d
1e
.
.
.
5x
5y
5z
```

while the expression

```
every Reverse{write, 1 to 5, !&lcase}
```

produces the output

```
1a
2a
3a
4a
5a
.
.
.
1z
2z
3z
4z
5z
```

Both Lifo and Reverse produce all possible combinations of their arguments. However, the expression

```
every Parallel{write, 1 to 5, !&lcase}
```

produces the output

```
1a
```

3.3.3 Alternate Methods of Parallel Resumption

The parallel resumption regime shown earlier terminates when any argument expression is depleted, that is, it does not produce another result.

An alternate method of parallel evaluation is to use the last result produced by each argument expression once that expression is depleted.

```
procedure Allpar(a)
   local i, x, done
   x := list(*a)
   done := list(*a, 1)
   every i := 1 to *a do x[i] := @a[i] | fail
   repeat {
      suspend Call(x)
      every i := 1 to *a do
         if done[i] = 1 then ((x[i] := @a[i]) | (done[i] := 0))
      if not(!done = 1) then fail
       }
end
```

This regime terminates when none of the argument expressions produces a result. For example,

```
every Allpar{|write \ 5, "a" | "b", "a" | "b" | "c"}
```

produces the output

```
aa
bb
bc
bc
bc
```

The first argument is |write in order to produce the result write repeatedly. Again, this argument could be treated differently from the others to avoid this effect.

A third approach to parallel evaluation is to evaluate each expression anew when it fails to produce a result. Again, this regime fails when none of the argument expressions produces a result.

```
procedure Rotate(a)
   local i, x, done
   x := list(*a)
   done := list(*a, 1)
   every i := 1 to *a do x[i] := @a[i] | fail
   repeat {
      suspend Call(x)
      every i := 1 to *a do
         if not(x[i] := @a[i]) then {
            done[i] := 0
            if !done = 1 then {
               a[i] := ^a[i]
               x[i] := @a[i] | fail
               }
            else fail
            }
         }
end
```

For example,

```
Rotate{|write \ 10, "a" | "b", "a" | "b" | "c"}
```

produces the output

```
aa
bb
ac
ba
ab
bc
aa
bb
ac
ba
```

Since |write is limited to ten results, the result sequence for the expression above is limited to the concatenation of the result sequences of ten procedure invocations.

3.3.4 Extracting Results

The evaluation regimes presented thus far treat the first argument expression as a procedure and the rest as arguments to this procedure. This interpretation of argument expressions is not inherent to argument evaluation regimes. An example of a regime with a different form follows.

This regime uses the second expression as the procedure and the rest of the even-numbered expressions as arguments to the procedure. Each odd-numbered expression produces a sequence of positive integers. The *i*th result of each even-numbered expression is used if and only if *i* is in the sequence produced by the odd-numbered expression preceding it.

```
procedure Extract(a)
   local i, j, n, x
   x := list(*a/2)
   repeat {
      i := 1
      while i < *a do {
         n := @a[i] | fail
         every 1 to n do
            x[(i + 1)/2] := @a[i + 1] | fail
         a[i + 1] := ^a[i + 1]
         i +:= 2
         }
      suspend Call(x)
      }
end
```

For example,

```
every Extract{3, |write, 2, "a" | "b" | "c", 3, "a" |"b" | "c"}
```

produces the output

```
bc
```

while

```
every Extract{1 to 3, |write, 1 | 3, "a" | "b" | "c", 3 | 1, "a" | "b" | "c"}
```

produces the output

```
ac
ca
```

Note that the expression

```
every Extract{3, |write, 2, "a" | "b" | "c", 5, "a" | "b" | "c"}
```

does not produce any result.

Some of the previous regimes can be simulated with **Extract**. For this purpose, it is useful to have a procedure such as the following:

```
procedure int(i)
   suspend i | |(i +:= 1)
end
```

which generates the integer sequence

```
i
i + 1
i + 2
  .
  .
  .
```

For example,

```
every write(int(5))
```

produces the output

```
5
6
7
8
9
10
 .
 .
 .
```

Using int,

```
Parallel{e1, e2, e3}
```

can be modeled with

```
Extract{int(1), e1, int(1), e2, int(1), e3}
```

4. Programmer-Defined Invocation

4.1 Introduction

Programmer-defined evaluation regimes provide a way to control the result sequence produced by a sequence of procedures and argument expressions. However, each procedure invocation is handled by the built-in Icon procedure evaluation mechanism. This is done using the procedure Call (See Section 3.1). There is no way of controlling the result sequence of a single procedure invocation within the regime. For example,

```
Parallel{find | match, |"a", |"and it got dark"}
```

produces the concatenation of the result sequences for

```
find("a", "and it got dark")
match("a", "and it got dark")
```

Therefore, its result sequence is {1, 13, 2}.

Suppose, however, that it is desirable to produce only one result from the invocation of find, thus producing the result sequence {1, 2}. There is no direct way to do this in Icon, although a procedure of the form

```
procedure find1(x, y)
   return find(x, y)
end
```

could be written. The evaluation regime then could be called as

```
Parallel{find1 | match, |"a", |"and it got dark"}
```

This approach is awkward, since a procedure such as find1 would have to be written for each procedure that is to be limited. Not only is this time consuming, but it is impractical if the procedure that is to be used is computed rather than being given explicitly.

It would be more desirable to be able to make a call such as

Parallel{find \ 1 | match, |"a", |"and it got dark"}

This does not work as intended, however, since it causes the following procedure invocation

(find \ 1)("a", "and it got dark")

which is equivalent to

find("a", "and it got dark")

while what is actually desired is

find("a", "and it got dark") \ 1

This need for control over procedure invocation motivates the following programmer-defined invocation regime facility

4.2 Implementation

A programmer-defined invocation regime is represented by a record of type Pdir with the following declaration

record Pdir(R, P, A)

where R is a invocation regime, P is a procedure that R acts on, and A is a list of any additional arguments used by R A list is used to pass the additional arguments so that a programmer-defined invocation regime may have an arbitrary number of arguments

The *i*th argument of the list A in a Pdir x is referenced by

x A[i]

A Pdir is used in the form

P{Pdir, $expr_2$,   , $expr_n$}

in place of

P{$expr_1$, $expr_2$,   , $expr_n$}

as in Section 3 1 The following procedure is used in place of Call to perform procedure evaluation

```
procedure Evalp(a)
   local lim
   case type(a[1]) of {
      "Pdir" : suspend a[1] R(a)
      "procedure" : suspend Call(a)
      }
end
```

If a Pdir is used, Evalp invokes the programmer-defined invocation regime Note that a[i] is a record of type Pdir and the programmer-defined invocation regime is the R field of this record The invocation regime is called with the list of argument results from the evaluation regime so that it can perform a procedure invocation If a programmer-defined invocation regime is not used, Evalp merely invokes Call as before

4.3 Examples

4.3.1 Limiting the Size of a Result Sequence

The limitation of the result sequence of find in the call

Parallel{find | match, |"a", |"and it got dark"}

can now be accomplished easily by writing a general-purpose limitation regime

```
procedure flim(a)
   suspend Call([a[1].P] ||| a[2:0]) \ a[1].A[1]
end
```

Note that the entire list of argument results produced by the evaluation regime is passed to the invocation regime. This is necessary, since the invocation regime uses Call to invoke the procedure. The expression

```
[a[1].P] ||| a[2:0]
```

concatenates the procedure to be invoked and the rest of the argument results produced by the evaluation regime to produce a single list as the argument for Call. For example, consider the expression

```
find1 := Pdir(flim, find,[1])
```

If

```
a := [find1, "a", "and it got dark"]
```

then Evalp invokes flim(a) (see Section 4.2) and

```
[find] ||| ["a", "and it got dark"]
```

produces the list

```
[find, "a", "and it got dark"]
```

which is passed to Call. This causes flim to suspend with

```
Call([find, "a", "and it got dark"]) \ 1
```

since, in this case, a[1].A[1] produces 1.

For example, to limit the result sequence of find to one result in the preceding invocation of Parallel, the invocation is replaced with

```
Parallel{find1 | match, |"a", |"and it got dark"}
```

which produces the results of

```
find("a", "and it got dark") \  1
match("a", "and it got dark")
```

An area that holds more possibilities for this invocation regime is text editing. For example, given a list of strings, it may be useful to find the first occurrence of each string in a piece of text, so that these strings can later be replaced or altered. The following expression finds the position where each lowercase letter first occurs in text.

```
Parallel{|find1, !&lcase, |text}
```

The result sequence for this invocation is the concatenation of the result sequences for

```
find("a", text) \ 1
find("b", text) \ 1
                :
find("z", text) \ 1
```

For example, the following segment of program

```
text := "Look for letters in this sentence"
every i := Parallel{|find1, !&lcase, |text} do
   write(text[i], " ", i)
```

produces the output

```
c 32
e 11
f 6
h 22
i 18
k 4
l 10
n 19
o 2
r 8
s 16
t 12
```

Note that the following segment of program produces the same output as the one above.

```
text := "Look for letters in this sentence"
every i := Lifo{find1, !&lcase, text} do
   write(text[i], " ", i)
```

4.3.2 Selection

In Section 2.2 a control operation Select was introduced to allow the programmer to select specific results from a result sequence. A programmer-defined invocation regime that provides this same ability within an evaluation regime is

```
procedure fsel(a)
   suspend Select{Call([a[1].P] ||| a[2:0]), a[1].A[1])}
end
```

Note again that a[1].P produces the procedure to be invoked, a[1].A[1] produces a positive integer, and the results in a[2:0] are arguments for the procedure to be invoked.

Consider the expression

```
Parallel{find | match, "a" | "b", |"bbbaaa"}
```

which produces the result sequence {4, 5, 6, 2}, the concatenation of the result sequences for

```
find("a", "bbbaaa")
match("b", "bbbaaa")
```

To produce only the second result produced by the procedure call

```
find("a", "bbbaaa")
```

the expression above is changed to

```
Parallel{Pdir(fsel, find,[2]) | match, "a" | "b", |"bbbaaa"}
```

This expression produces a result sequence that is the concatenation of the result sequences for

```
Select{find("a", "bbbaaa"), 2}
match("b", "bbbaaa")
```

namely {5, 2}.

Note that the expression

```
Lifo{find, "a", "aabb"}
```

produces the result sequence for

```
find("a", "aabb")
```

which is {1, 2}. It might seem that

```
Lifo{Pdir(fsel, find,[1 | 2]), "a", "aabb"}
```

would produce the result sequence for

```
Select{find("a", "aabb"), 1 | 2}
```

Since the PDCO facility creates co-expressions for all the arguments of Lifo, the result sequence produced is actually the concatenation of the result sequences for

```
Select{find("a", "aabb"), 1}
Select{find("a", "aabb"), 2}
```

namely {1, 2}, as expected. On the other hand, consider the expression

```
Parallel{find | match, "a" | "b", |"bbbaaa"}
```

The following segment of program

```
Parallel{Pdir(fsel, find,[2 | 3]) | match, "a" | "b", |"bbbaaa"}
```

might be used to find only the second and third results of the call

```
find("a","bbbaaa")
```

and all the results of

```
match("b", "bbbaaa")
```

Because of the way co-expressions are handled in Icon, the following sets of triples are produced:

```
{Pdir(fsel, find, [2]), "a", "bbbaaa"}
{Pdir(fsel, find, [3]), "b", "bbbaaa"}
```

Note that match is never invoked, since the second argument only produces two results. These triples produce the concatenation of the result sequences for

```
Select{find("a", "bbbaaa"), 2}
Select{find("b", "bbbaaa"), 3}
```

which is {5, 3}, while the desired result sequence is {5, 6, 2}. To produce the desired result sequence,

```
Parallel{Pdir(fsel, find,[2 | 3]) | match, "a" | "a" | "b", |"bbbaaa"}
```

must be used. The result sequence of this expression is the concatenation of the result sequences for

```
Select{find("a", "bbbaaa"), 2}
Select{find("a", "bbbaaa"), 3}
match("b", "bbbaaa")
```

4.3.3 Echoing Procedure Invocation

Sometimes it is useful to see what procedure invocations would result from an evaluation regime, without actually causing these invocations. This can be done with the following invocation regime.

```
procedure Echo(a)
   local str, i
   str := (image(a[1].P))[upto(' ', image(a[1].P)) + 1 : 0] || "("
   every i := !a[2:0] do  {
      if type(i) == "string" then
         i := "\"" || i || "\""
      str := str || i || ","
      }
   str[-1] := ")"
   suspend str
end
```

For example, the program segment

```
text := "Look for letters in this sentence"
every write(Parallel{|Pdir(Echo, find), !&lcase, |text})
```

produces the output

```
find("a", "Look for letters in this sentence")
find("b", "Look for letters in this sentence")
find("c", "Look for letters in this sentence")
                :
find("z", "Look for letters in this sentence")
```

which is the procedure invocations caused by

```
text := "Look for letters in this sentence"
every write(Parallel{find, !&lcase, |text})
```

4.3.4 Positive Result Selection

Consider a procedure Roots(a, b, c) that returns the real roots of the quadratic equation ax^2+bx+c For example, Roots(1, 2, 1) has the result sequence {−1,−1}, Roots(1, −1, −2) has the result sequence {2, −1}, while Roots(1, 1, 4) has an empty result sequence, since the quadratic equation with these coefficients has no real roots. The segment of program

```
b := [0, -3, 1, 0, 2]
c := [-1, 2, -1, 0, 0]
every write(Parallel{Roots, |1, !b, !c})
```

produces the following output:

```
1.0
-1.0
2.0
1.0
0.61803399
-1.618034
0.0
0.0
-2.7755576e-17
-2.0
```

Suppose that only the positive real roots are desired. An invocation regime to do this is

```
procedure Positv(a)
   local i, label
   if *a[2:0] > 0 then {
      label := ""
      every i := !a[2:0] do label := label || i || "  "
      suspend label
      }
   every i := Call([a[1].P] ||| a[2:0]) do
      if i > 0 then suspend i
   write()
end
```

The appearance of the previous output makes it impossible to tell which roots go with which quadratic equation. This invocation regime also echoes the arguments to the procedure, in a manner similar to that of Echo. Now, the segment of program

```
b := [0, -3, 1, 0, 2]
c := [-1, 2, -1, 0, 0]
every write(Parallel{|Pdir(Positv, Roots), |1, !b, !c})
```

produces the output

```
1  0  -1
1.0

1  -3  2
2.0
1.0

1  1  -1
0.61803399

1  0  0

1  2  0
```

Note that this invocation regime differs from the previous invocation regimes introduced in several respects: (1) the value of each result, rather than its position in the result sequence of a procedure, determines whether or not it is selected, and (2) extra information (labeling) is produced for ease of reading. Note that the only way to produce these labels without a programmer-defined invocation regime is to actually rewrite Roots.

5. Conclusions

Programmer-defined control operations have proved useful in two ways: (1) programming situations in which the features of Icon are inadequate and (2) for providing insights into the interaction of generators and sequencing of expression evaluation [5]. Programmer-defined evaluation is a subset of PDCO that has proven particularly useful in both ways.

There are two major factors that make programmer-defined evaluation useful. First, the ability of expressions to produce more than a single result makes it possible for a procedure call to produce more than one procedure activation. Each activation may use a different n-tuple of arguments. Second, a sequence of procedure activations results from each procedure call. This sequence is determined by the built-in Icon evaluation regime. Programmer-defined evaluation regimes allow this built-in evaluation regime to be studied by comparing it to other regimes that can be developed. Since some of these regimes produce different sequences of procedure activations than the built-in regime, they often are useful as programming tools.

A PDCO facility could be added to any programming language in which expressions can be treated as data objects [5]. This also applied to programmer-defined evaluation regimes. As with PDCO in general, the usefulness of programmer-defined evaluation regimes in other languages is limited. Since most programming languages allow an expression to produce at most one result, a procedure call results in at most one procedure activation. It is generators that provide a wide variety of possible procedure activations in Icon.

Programmer-defined evaluation provides insights into the built-in Icon procedure evaluation mechanism, as well as allowing alternatives to this evaluation mechanism to be explored. The usefulness of some of these evaluation regimes, such as parallel resumption, provides a convincing argument for elevating some underlying mechanisms in many languages to the source-language level.

Acknowledgment

The authors are indebted to Steve Wampler for a number of helpful suggestions on the presentation of the material in this paper.

References

1. Fisher, D. A. *Control Structures for Programming Languages*, Ph.D. dissertation, Computer Science Department, Carnegie-Mellon University (1970).
2. Coutant, C. A., Griswold R. E., and Wampler, S. B. *Reference Manual for the Icon Programming Language; Version 5 (C Implementation for UNIX)*, Technical Report TR 81-4a, Department of Computer Science, The University of Arizona (1981).
3. Wampler, S. B. *Control Mechanisms for Generators in Icon.* Ph.D. dissertation, Department of Computer Science, The University of Arizona (1981).
4. Wampler, S. B. and Griswold, R. E. "Result Sequences", *Computer Languages*, Vol. 8, No. 1 (1983). pp. 1-14.
5. Griswold, R. E. and Novak, M. "Programmer-Defined Control Operations in Icon", *The Computer Journal*, Vol. 26, No. 2 (May 1983). pp. 175-183.
6. Wampler, S. B. and Griswold, R. E. "Co-Expressions in Icon", *The Computer Journal*, Vol. 26, No. 1 (February 1983). pp. 72-78.