The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1995
Number 28

In this issue ...

String Invocation

String invocation allows an operation to be invoked by its string name. In Icon, the term operation includes functions like write(), operators like +, declared procedures, and record constructors.

Being able to invoke an operation by its name allows the operations that are to be performed to be determined when a program is run. With string invocation, for example, it is possible for a user to specify, via input to a program, a particular function to be evaluated, even if the function does not occur explicitly in the program.

This article describes string invocation, starting with a review of what's involved in the invocation process.

Invocation

In Icon terminology, functions are built-in, while procedures are declared. From the point of view of invocation, there is no difference between what's built in and what's declared. In this article, we'll generally use the term procedure for both, and only point out the difference in the few cases where that's necessary.

There are two ways a procedure can be invoked (called): with explicit arguments, as in

```
write(heading, " ", text)
```

or with the arguments given in an Icon list, as in

```
write ! values
```

where values contains the strings to be written. (In Version 9 of Icon, this form of invocation can be used with records as well as lists; we'll just refer to lists in what follows.)

The main advantage of invocation with an Icon list is that it allows the number of arguments to be determined when the program is run. This is useful primarily for built-in functions like write() and DrawLine() that accept a variable number of arguments. A declared procedure also can have a variable number of arguments.

Although the procedure to be invoked usually is given explicitly in an invocation expression, it can be the result of an expression that produces a procedure, as in

```
(if \eol then write else writes)(greeting)
```

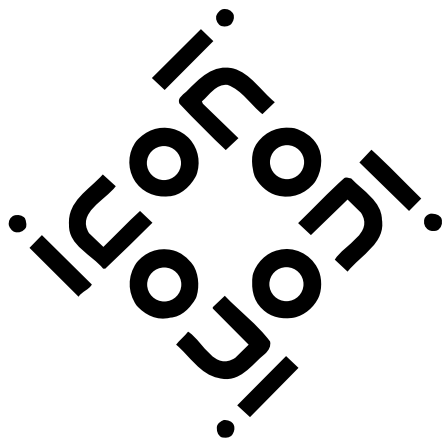
in which the if–then–else expression produces either write or writes. Depending on the value of eol, the result is either

```
write(greeting)
```

or

```
writes(greeting)
```

In order to make the description of invocation general, we'll use the forms



expr0(*expr1*, ..., *exprn*)

and

expr0 ! *expr1*

where *expr0*, *expr1*, ..., *exprn* all can be arbitrarily complicated expressions.

In most invocation expressions, *expr0* is a variable whose value is a procedure. A variable whose name corresponds to a procedure is global by default and has that procedure as its initial value. We'll refer to this as an explicit reference to the procedure. It's important to remember that procedures are data values in Icon. For example,

```
type(write)
```

produces "procedure", the type procedure including both built-in functions like write() and declared procedures like main().

As we've said before [1], it's possible to replace the initial value of a variable that corresponds to a procedure by some other value — even a value that's not a procedure. This has its uses, but if it's done accidentally, the results can be disastrous. For this article, we'll assume that variables with names that correspond to procedures retain their initial values.

In the expressions

expr0(*expr1*, ..., *exprn*)

and

expr0 ! *expr1*

evaluation is strictly left to right. Although *expr0* usually is a variable for which evaluation can't fail, if *expr0* is an expression that fails, evaluation stops at that point and the entire invocation expression fails. That's logical; there's nothing to invoke.

If *expr0* succeeds, *expr1*, ..., *exprn* are evaluated before anything else is done. If any of these expressions fails, control backtracks to the previous expression for a possible alternative that might make the subsequent expression succeed [2]. In any case, all argument expressions must eventually succeed; otherwise the entire invocation expression fails because there is not a complete set of arguments.

Once all of *expr0*, *expr1*, ..., *exprn* are evaluated successfully, the type of *expr0* is checked, There are three acceptable possibilities for the *value* of *expr0*:

1. A procedure.

2. An integer or a type that can be converted to an integer.

3. A string that is the name of an operation.

The first case is the most common. The second case, called mutual evaluation, doesn't invoke an operation. Instead, it produces the result of the argument in the position given by *expr0*. For example,

```
2(expr1, expr2, expr3)
```

produces the result of *expr2*, assuming all the argument expressions succeed.

The third case leads us to the main subject of this article.

Names for Operations

There are string names for all operations in Icon (with one exception that we'll mention later). The string name of an operation can be used to invoke that operation.

The string name for a procedure is what you'd expect. For example, the name for write is "write". This means that you can use

```
"write"(heading, " ", text)
```

or

```
"write" ! values
```

and get the same results as if you'd used

```
write(heading, " ", text)
```

or

```
write ! values
```

There's not much purpose in using string invocation in the manner above, unless, perhaps, you want to see if it works. String invocation even is a little slower than using a procedure-valued variable, since the string "write" must be looked up in a table to find the corresponding procedure.

On the other hand, if the procedure to be invoked is not known when the program is written, string invocation can be very useful. We'll give some practical examples later, but for now, think about the possibilities of

```
read() ! values
```

Although the string names for functions are straightforward, the names for operators are not. The string names for operators are derived from

the characters used to represent them in ordinary situations. For example, "||" is the name for concatenation. Used in string invocation, such an operator must be cast in one of the invocation forms, as in

```
"||"(s1, s2)
```

or

```
"||" ! spair
```

One problem with the string names for operators is that some operator symbols are used for both unary (prefix) operations and binary (infix) operations. For example, * in prefix form produces the size of its operand, while * in infix form produces the product of its operands.

This ambiguity is resolved by the number of arguments given when the operator is used in string invocation. For example

```
"*"(x)
```

produces the size of x, while

```
"*"(i1, i2)
```

produces the product of i1 and i2.

In retrospect, it probably would have been better to have the number of arguments included in the string name of such operators, as in "*1" and "*2". It's too late now.

The names for operators that do not appear in prefix or infix form are derived from their syntactic appearances:

operation	name
x[i]	"[ ]"
x[i:j]	"[:]"
i to j by k	"..."

For example,

```
"..."(1, 10, 1)
```

generates the integers 1, 2, ..., 10. The third argument must be given explicitly; otherwise string invocation looks for a binary operator named "..." and doesn't find it.

There are no string names for x[i+:j] and x[i–:j] — these syntactic forms are shorthand for x[i:j] that the Icon translator handles; they are not distinct from x[i:j].

We mentioned earlier that there is one operation that does not have a string name and hence cannot be used in string invocation. It's explicit list construction, as in

```
[x1, x2, ..., xn]
```

There is no conceptual problem with string invocation for list construction, and the obvious string name is "[...]". The problem, instead, lies in the implementation of Icon. For technical reasons, the problem is hard to fix (it has to do with the fact that list construction is the only operation in Icon that takes an arbitrary number of arguments). On the other hand, this exception apparently has gone unnoticed until this article was written. (Analyst articles serve several purposes; one of the less obvious ones is that they illuminate the darkest corners of Icon.)

There is one problem with using string invocation for operators that needs mentioning. Although there are string names for assignment operators, there's no way to use them effectively with the interpreter. The problem is that an invocation expression deferences all its operands before the operation is applied. Consequently,

```
":="(x, 2)
```

does not assign 2 to x. Instead, x is dereferenced before the assignment is attempted and a run-time error results. This also is an implementation problem in the interpreter that is not easily fixed. The Icon compiler, however, does handle string invocation of assignment properly.

String Invocation and Linking

Starting with Version 9 of Icon, the linker, by default, discards declarations that are not explicitly referenced in a program. For example, if a program includes a procedure declaration, but there is no explicit reference to the procedure in the program, the procedure declaration is discarded by the linker. This eliminates unneeded code in the icode files that the linker produces. This optimization is particularly useful when a program links procedure libraries but does not use all the procedures in them.

This linker optimization is, however, a problem with the string invocation of declared procedures, since if there is no specific reference to the procedure in the program, the linker deletes it. If there is then an attempt to invoke the deleted procedure by its string name, a run-time error occurs. The problem may appear mysterious when an examination of the program shows the declaration of the procedure.

There's an easy way to avoid this kind of problem; just add

```
invocable all
```

to the program. This declaration prevents the linker from eliminating declarations that aren't specifically referenced. The invocable declaration is only needed for the string invocation of declared procedure and record constructors when using the Icon interpreter. When using the Icon compiler, it also is needed for built-in functions and operators that are invoked by their string names.

The declaration

```
invocable all
```

prevents the elimination of all unreferenced declarations. You can be more specific if you like, as in

```
invocable print, dump
```

which only prevents the elimination of declarations for print and dump.

Considering the mysterious effects that may occur in an attempt to invoke a procedure that's been deleted by the linker, it's generally better to use

```
invocable all
```

in programs that use string invocation.

Next Time

In a follow-up article, we'll show some examples of situations in which string invocation is particularly useful. We'll also have an article on a closely related subject — getting actual values for operations from their string names.

References

1. "Programming Tips", Icon Analyst 25, pp. 10-12.

2. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 81-83.

Random Number Generators

In Issue 26 of the Analyst, we described the use of random numbers in Icon. In this article, we'll discuss how sequences of random numbers are generated and, in particular, the method used by Icon. Much of the material here is based on Don Knuth's excellent and entertaining chapter on the subject in his series *The Art Of Computer Programming* [1]. If you haven't read his discussion of random number generation, we strongly recommend that you do. If you have, but not recently, you may find, as we did, much to learn on re-reading.

Background

Until the advent of computers, random selection was done using various physical methods, such as flipping coins or drawing balls out of an urn. Many elaborate and somewhat fanciful machines have been developed for this purpose. These machines are still used in lotteries — a spinning cage and dropping balls are much more impressive that having a computer pick numbers.

Once computers became available, mathematical methods of producing "random" numbers became a subject of much study. Of course, numbers picked by algorithmic means aren't truly random, but they can display the statistical properties of randomness well enough for most purposes.

John von Neumann was a pioneer in the area of computer generated random sequences. He suggested producing a sequence of numbers with apparent randomness by squaring the previous number in the sequence and extracting the middle digits of the result to produce the next number. Although this method looks good on the surface, the results actually aren't very random. Many other algorithms for producing random numbers have been proposed. On the other hand, there has been a tendency to use "hand-me-down" random number generators without critical thought, resulting in a fairly widespread use of defective random-number generators. The history of these subjects makes fascinating reading. Again, we refer you to Reference 1.

There are elaborate (and computationally expensive) methods of generating random numbers that are quite good. A simple and computationally inexpensive method works well enough for most purposes, however. A common method, and the one used by Icon, is called the linear conguential method. In this method, numbers in sequence are computed by the recurrence

$$n_{i+1} = (a * n_i + c) \bmod m$$

where there is an initial integer value, n_0, called the *seed*, and *a*, *c*, and *m* are parameters of the recurrence.

Configuring the Linear Congruential Method

The values of the parameters in the recurrence have an important effect on the quality of the numbers generated. Knuth covers this in some detail and then summarizes as follows:

1. n_0 can be chosen arbitrarily. Different seeds produce different starting places in the sequence.

2. The modulus *m* should be large. It typically is taken as the computer's word size and hence is a power of 2.

3. Assuming that *m* is a power of 2, the multiplier *a* should satisfy the relationship $a \bmod 8 = 5$. It also should be in the range

$$(m - \sqrt{m}) > a > m / 100$$

Finally, the digits of *a* should be chosen in a haphazard manner.

4. The additive constant *c* should be relatively prime to *m* (odd if *m* is a power of 2). Also, c / m should be approximately

$$(1/2) - (1/6)\sqrt{3} \approx 0.21132486 \ldots$$

5. When basing a decision on n_i, its most significant digits should be used, since the least significant ones are not particularly random.

The correct choices for *a* and *c* assure that the random number has period *m*; that is, that it runs through all integers from 0 to $m - 1$ before repeating.

As mentioned earlier, when a random number generator is needed, it is typical to borrow a random number generator from an existing application without critical consideration. In the case of the linear congruential method, the parameters usually are borrowed as well — without verifying that they meet the specifications above.

Icon's Random-Number Generator

Icon's random number generator was written long ago and no one remembers the reasons for the decisions made at the time, except that it was borrowed from another application and the parameters were changed somewhat.

For Icon's random number generator, the parameters are:

$n_0 = 0$ (the seed is given by &random)

$m = 2^{31}$ (the sign bit is omitted in 32-bit words for computational reasons)

$a = 1103515245$

$c = 453816694$

These values meet all the specifications listed above except for *c*, which should be odd. This is the reason why an even seed produces a sequence in which all the values are even, as noted in Issue 26 of the Analyst. Similarly an odd seed produces a sequence in which all the values are odd. In effect, this mistake in picking *c* splits Icon's random number generator into two distinct generators, each with a period of 2^{30}.

How serious is this? It probably isn't very serious. Each of Icon's two random number generator has half the period intended, but the periods still are very large, and this shouldn't be a problem in ordinary applications. In addition, each of the two sequences satisfies the specifications and each seems to have good statistical properties, at least using simple methods of testing. And there are potential uses for two independent random number generators.

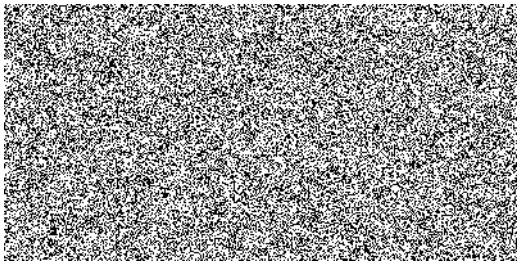
In any event, it's too late to "fix" the problem. A change now would have mysterious effects on some existing applications and invalidate test data that depends on the reproducibility of random-number sequences from run to run.

Evaluating Icon's Random Number Generator

We frequently hear complaints that Icon's random-number generator doesn't produce really random sequences. Any really random sequence is going to have runs that appear to be non-random. Are such complaints therefore perception problems, or is there a real problem?

There are good ways of testing a random number generators, such as the spectral analysis test. We don't have a package to do this and implementing one is more work than we're willing to undertake. We'd welcome help from someone with more knowledge and facilities than we have.

One simple method of testing a random number generator is to use it to pick points and plot them within a rectangular area. Here's a result of doing this for one of Icon's random-number generators (the other one produces similar results):



This is not bad as random-number generators go — some random number generators display pronounced bands in which there are no values or in which the density of values is much higher than other areas. Of course, you have to take our word for the fact that such an image develops relatively uniformly over time.

Rolling Your Own

It's easy to experiment with different parameters by writing a linear congruential random number generator in Icon. Here's a procedure that you can use; it defaults to Icon's built-in parameters as shown. The global variable random serves the role of &random in built-in generation.

```
global random

procedure rand_num(a_, c_, m_)
  static random_last, a, c, m

  initial {
    /random := 0
    a := \a_ | 1103515245
    c := \c_ | 453816694
    m := \m_ | 2 ^ 31
    }

   return random := (a * random + c) % m

end
```

Here's a procedure that does what Icon's built-in ?i operation does. It shows how a value in a range is selected from a number in the random sequence.


```
procedure rand_int(i)
  static scale

  initial scale := 1.0 / (2 ^ 31 – 1)

  (i := (0 < integer(i))) | runerr(205, i)

  return integer(i * rand_num() * scale) + 1

end
```

Reference

1. Donald E. Knuth, *Seminumerical Algorithms, The Art of Computer Programming*, Vol. 2, Addison-Wesley, 1969, pp. 1-160.

Lindenmayer Systems

So far, we've looked at implementing L-systems by drawing while repeatedly processing the symbols in the replacement rules. This is an interpretive method. Interpretation is natural for L-Systems, and, as is the case for interpreters in general, it is easy to implement and modify.

An alternative approach is to compile an L-system into a program, which when run, produces the corresponding drawing. The target language for the compiler could be any language that supports turtle graphics; Icon is the obvious choice.

It's worth noting that one of the main motivations for compiling rather than interpreting is speed. That's not a major issue here — drawing L-systems hardly classifies as a production job for which the maximum execution speed is essential.

As is generally the case, writing a compiler involves a level of indirection — writing a program to write a program. Before starting, it's necessary to have a model for the code to be generated. We'll look at that first and then go on to the compiler itself. For simplicity, we'll use the version of L-systems that use single-character symbols.

In the code produced by the compiler, each symbol for which there is a replacement in the L-system is translated into a procedure, which, when called, performs the drawing for that symbol.

Consider this L-System, which we've used in previous articles:

```
X–>F–[[X]+X]+F[+FX]–X
F–>FF
axiom:X
angle:22.5
gener:5
```

The procedure for the axiom X illustrates the code to be produced:

```
procedure X(gener)
  if gener > 0 then {
    F(gener – 1)          # F
    TLeft(Angle)          # –
    TSave()               # [
    TSave()               # [
    X(gener – 1)          # X
    TRestore()            # ]
    TRight(Angle)         # +
    X(gener – 1)          # X
    TRestore()            # ]
    TRight(Angle)         # +
    F(gener – 1)          # F
    TSave()               # [
    TRight(Angle)         # +
    F(gener – 1)          # F
    X(gener – 1)          # X
    TRestore()            # ]
    TLeft(Angle)          # –
    X(gener – 1)          # X
    }
  return
end
```

The argument gener is the number of generations remaining. When the number of remaining generations reaches 0, the procedure simply returns. Otherwise, it calls turtle graphics procedures for drawing or other procedures produced by the compiler for symbols that have replacement rules.

The treatment of symbols corresponding to drawing actions illustrates an interesting aspect of L-systems. There's an unwritten rule that replacements cannot be specified for +, –, [, and], and they are not translated into drawing procedures. However, as illustrated by our example L-system, replacements can be specified for F. This may seem paradoxical, but it's the essence of L-systems, in which at every generation but the last, what would be a line is replaced by a more complicated drawing. The same is true of f (which skips without drawing), although f does not occur in our example.

Handling F and f requires special cases, since when their procedures reach the last generation, a line is drawn. For our example L-system, the procedure for F is:

```
procedure F(gener)
  if gener > 0 then {
    F(gener – 1)          # F
    F(gener – 1)          # F
    }
  else TDraw(Length)      # F
  return
end
```

Writing an Icon program to convert an L-system into a corresponding program is relatively straightforward:

```
$define Indent 3        # code indentation

procedure main()
  local line, sym, new, keyword, value
  local axiom, gener, angle, length
  local allsyms, replace, procs

  allsyms := ''         # initially empty
  procs := table()      # procedure symbols

  gener := 4            # defaults
  length := 5
  angle := 90.0

  while line := read() do
    line ? {
      if sym := tab(find("–>")) then {
        move(2)
        replace := tab(0)
        procs[sym] := replace
        allsyms ++:= replace
        }
      else if keyword := tab(find(":")) then {
        move(1)
        value := tab(0)
        case keyword of {
          "axiom":   {
            axiom := value
            allsyms ++:= value
            }
          "gener": gener := integer(value) |
              stop("*** invalid generation_
                specification")
          "angle": angle := real(value) |
              stop("*** invalid angle: ", line)
          "length": length := integer(value) |
              stop("*** invalid length: ", line)
          default:  stop("*** invalid keyword: ",
            line)
          }
        }
      else stop("*** invalid specification: ", line)
      }

  # Be sure a procedure is produced for all
  # symbols used, even if not defined, except
  # for +, –, [, and ].
```

```
   allsyms --:= '+-[]'

   every sym := !allsyms do
      /procs[sym] := ""        # empty replacement

   # Write heading and main procedure.

   write("link turtle")
   write()
   write("$define Generations ", gener)
   write("$define Angle ", angle)
   write("$define Length ", length)
   write()
   write("procedure main()")
   gencode(axiom, "Generations", Indent)
   write("end")
   write()

   # Produce drawing procedures.

   every sym := key(procs) do
      genproc(sym, procs[sym])
end

procedure gencode(replace, arg, indent)
   local sym, pad

   pad := repl(" ", indent)

   every sym := !replace do {
      case sym of {
         "+":      write(pad, "TRight(Angle)   #+")
         "-":      write(pad, "TLeft(Angle)    #-")
         "[":      write(pad, "TSave()         # [")
         "]":      write(pad, "TRestore()      # ]")
         default: write(pad, sym, "(", arg, ") # ",
            sym)
         }
      }

   return

end

procedure genproc(name, replace)

   write("procedure ", name, "(gener)")
   write("   if gener > 0 then {")
   gencode(replace, "gener - 1", 2 * Indent)
   write("      }")
   case name of {
      "F": write("   else TDraw(Length)      # F")
      "f": write("   else TSkip(Length)      # f")
      }
   write("   return")
   write("end")
   write()

   return

end
```

The first part of the compiler, which is similar to the first part of the interpreter, processes the L-system. No code is produced until the entire L-system is processed; this ensures that the output is in the proper order.

Once the L-system has been processed, code is produced: first a link declaration, then preprocessor definitions for the L-system parameters, a main procedure for the axiom, and finally procedures for the rules. The procedure gencode() produces code for the replacements, while the procedure genproc() produces the rest of the procedure declarations, including conditional code and the handling of F and f. That's all there is to it.

For the example L-system we've been using, the program produced is:

```
link turtle

$define Generations 5
$define Angle 22.5
$define Length 5

procedure main()
   X(Generations) # X
end

procedure X(gener)
   if gener > 0 then {
      F(gener - 1)          # F
      TLeft(Angle)          # -
      TSave()               # [
      TSave()               # [
      X(gener - 1)          # X
      TRestore()            # ]
      TRight(Angle)         # +
      X(gener - 1)          # X
      TRestore()            # ]
      TRight(Angle)         # +
      F(gener - 1)          # F
      TSave()               # [
      TRight(Angle)         # +
      F(gener - 1)          # F
      X(gener - 1)          # X
      TRestore()            # ]
      TLeft(Angle)          # -
      X(gener - 1)          # X
      }
   return
end

procedure F(gener)
   if gener > 0 then {
      F(gener - 1)          # F
      F(gener - 1)          # F
      }
```

```
      else TDraw(Length)     # F
      return
   end
```

An interesting exercise is to convert the compiler to handle multi-character symbols as described in the last issue of the Analyst.

Acknowledgment

Gregg Townsend suggested the idea of compiling L-systems and provided the model used here for code generation.

Dynamic Analysis of Icon Programs

Introduction

In the last issue of the Analyst, we described a method for analyzing the static properties of Icon programs and presented the results of static analysis of the Icon program library.

Static analysis is interesting and provides a measure of the relative frequency of occurrence of different features of Icon , but it sheds little light on how frequently features actually are used. For example, static analysis cannot tell how many times an expression is evaluated, if at all.

Static analysis also cannot tell if a function is what it appears to be — Icon allows values to be assigned to variables that initially have function values and allows function values to be assigned to other variables. With string invocation, as described in the first article of this issue of the Analyst, a function or operator can be invoked without ever appearing in the text of a program. Fortunately, such things are relatively rare and have little effect on the static analysis of most programs.

Dynamic analysis, on the other hand, potentially can determine how often any expression in a program is evaluated and provide many other kinds of information about program execution, such as storage allocation and automatic type conversion.

Obtaining Information During Program Execution

Unlike static analysis, in which information can be obtained by examining the text of the source program, dynamic analysis requires information about what goes on when a program is running. Obtaining this information is by far the hardest part of dynamic analysis. There are three main ways by which information about a running program can be obtained:

1. Manual instrumentation of the source program (SP).

2. Automatic instrumentation of the SP by a preprocessor.

3. Instrumentation of the implementation of the language itself.

Manual instrumentation consists of adding expressions to the program to produce information as the program runs. Such instrumentation

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U.S.A.

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...uunet!arizona!icon-project

THE UNIVERSITY OF
ARIZONA ®
TUCSON ARIZONA

and

Bright Forest Publishers

Tucson Arizona

can be as simple as write() expressions— something we all add to our programs from time to time. An alternative is calling procedures that accumulate information that is written out just before the program terminates.

There are several problems with manual instrumentation:

1. It is error prone — it is all too easy even with something as simple as a write() expression to mangle a loop and cause a program to malfunction.

2. Some kinds of information are difficult or impossible to get with manual instrumentation. Storage allocation is an example.

3. Extensive manual instrumentation is impracticably labor intensive.

4. Manual instrumentation may distort program behavior in subtle ways, even if only by increasing the size of the SP or slowing down its execution.

5. Manual instrumentation may have to be modified or redone if the SP is changed.

Using an instrumenting preprocessor avoids some of the problems with manual instrumentation. A correct preprocessor will not introduce errors into the SP and it easily can handle exhaustive instrumentation without modifying the SP itself. Instrumenting preprocessors cannot solve the problems of distorting program behavior, however; in fact, they generally make such problems worse. It's also difficult to provide selective instrumentation when using a preprocessor. And an instrumenting preprocessor cannot provide information that manual instrumentation cannot.

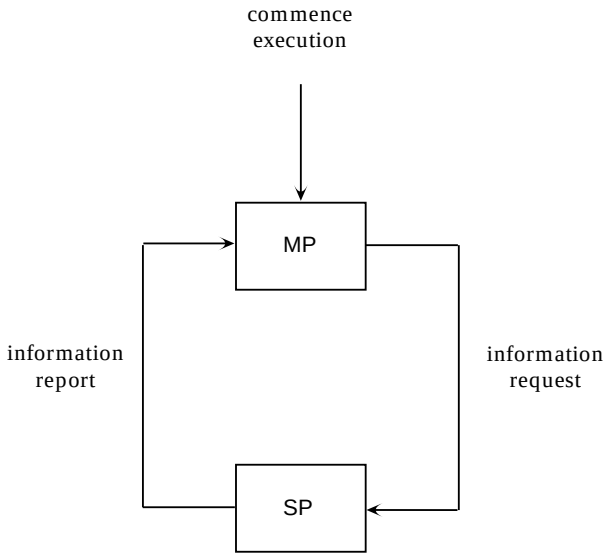
The variant translator that implements the instrumentation described in Issue 6 of the Analyst [1] illustrates the advantages and disadvantages of using a preprocessor. It provides exhaustive instrumentation of expression evaluation, but it transforms an SP into a much larger program — typically by a factor of 4 — and the resulting program runs much more slowly — typically by a factor of 25.

The third alternative, adding instrumentation to the implementation of Icon itself, can overcome all of the problems mentioned above. The SP need not be modified to produce run-time information, information can be provided about internal operations such as storage allocation, provisions can be made to provide selective information, the size of the SP is not affected, and the degradation in running speed of the SP in terms of its CPU time is minor.

In the case of an interpreter like Icon, the instrumentation can be added to the run-time system. This was done to obtain the information about storage management described in Issues 1 and 2 of the Analyst [2, 3]. Later the implementation was recast for use in MT Icon [4] and extended to cover many aspects of program execution [5].

In MT Icon, the SP and a monitoring program (MP) run in the same execution space, with the MP requesting the run-time system to provide information about the execution of the SP.

When the MP requests information about events that occur during the execution of the SP, the Icon run-time system turns control over to the SP until it detects a requested event. When that happens, execution in the SP is suspended and the MP is activated and provided with the requested information. The MP then performs whatever processing it needs to do and then requests information about SP execution again. Thus, control passes back and forth between the MP and the SP, with the MP in control. The SP is "unaware" that it is being monitored; the only difference it can detect is slower passage of wall clock time because it is sharing real time with the MP. The flow of control is shown in the following diagram:



In this article and ones to follow, this method will be used for the dynamic analysis of Icon programs.

The information requested about events in the SP and how this information is used depends

on the MP. An MP can request information about various kinds of events, write it out as it receives it, present it visually [6], or accumulate it and produce a summary report when execution of the SP terminates. We'll use summary reports in what follows, but we plan to have something to say about the visual presentation of program execution in a later article.

The structure of MPs that produce summary reports is described in Reference 5. We'll review the techniques here and show examples before going on the substance of dynamic analysis — what actually happens during program execution.

Writing Monitoring Programs

Writing an MP to collect and report information about aspects of the execution of an SP is not a difficult task. Most of what's needed is built into MT Icon or provided by procedures and definitions in the Icon program library. Here's the basic model for an MP:

```
link evinit                # support procedures
$include "evdefs.icn"      # event definitions
procedure main(args)
   EvInit(args)            # load the SP
   while EvGet() do {      # request event in SP
      …                    # process event
      }
   …                       # final processing
end
```

The procedures in evinit are needed to support loading and activation of the SP. The file evdefs.icn contains the definitions of various symbols that may be needed by the MP.

When an MP is called, the SP to be monitored and any arguments for it are given on the command line, as in

```
alloc rsg –l 1000 <rsg.dat
```

in which the MP alloc monitors the SP rsg with the command line option –l 1000 and standard input from the file rsg.dat.

The procedure EvInit() loads the SP, at which point monitoring can begin. The procedure EvGet() requests an event from the SP. When a event occurs in the SP, control is returned to the MP.

Two keywords are set for the MP's use when an event occurs in the SP: &eventcode, which identifies the kind of event that occurred, and &eventvalue, a value associated with the event. For example, in the case of allocation events, the event code indicates kind of storage allocated, such as for a string or a list, and the event value is the amount of allocation in bytes. Event codes are one-character strings. Event values depend on the type of event.

In the case of MPs that produce summary reports on completion of the SP, the event information can be accumulated in a table. When an event is processed, a request for another event is made. EvGet() fails when the SP terminates. At that point, any final processing, such as producing a summary report, can be done.

If EvGet() is called without an argument, any event in SP returns control to the MP. There are over 100 different kinds of events that can be reported. The most important event categories are:

category	number of event types
expression evaluation	16
structure access	29
string scanning	6
assignment	3
type conversion	3
storage allocation	21
garbage collection	4

Most MPs only are interested in certain types of events, such as storage allocation events. An argument, called an *event mask*, can be given in the call of EvInit() to limit monitoring to the events of interest.

For example, FncMask, defined in evdefs.icn, specifies events related to the evaluation of built-in functions. Thus, EvGet(FncMask) limits monitoring to the evaluation of built-in functions. The kinds of events associated with the evaluation of built-in functions are calls, returns, suspensions, and removal of suspended function calls that cannot be resumed. An example of removal occurs in

```
if find(s) then …
```

If find() suspends, the then clause is evaluated and the call of find() cannot be resumed.

Figure 1 on the next page shows an MP that accumulates and summarizes information about the evaluation of functions in an SP. The library file evnames contains a procedure that maps event codes to descriptive phrases. EvInit() fails if the

```
link evinit                          # monitoring initialization
link evnames                         # event-name mapping
$include "evdefs.icn"                # event definitions
procedure main(args)
  local summary, event
  EvInit(args) |                     # initialize interface
    stop("*** cannot load program")
  summary := table(0)                # table to accumulate events
  while EvGet(FncMask) do            # get function events
    summary[&eventcode] +:= 1        # tabulate them
  write(left("event", 30), right("count", 8), "\n")
  every event := !FncMask do         # list events
    write(left(evnames(event), 30), right(summary[event], 8))
end
```

Figure 1. A Monitor to Summarize Function Usage

specified program cannot be loaded; the MP terminates with an error message if this happens. The table summary is used to keep track of how many times each kind of function event occurs — that is, how many times a function is called and how many times the result is a return, suspension, failure, or removal.

When the SP terminates, the table summary is subscripted by the event codes in FncMask and the accumulated results are written out, using evnames() to map event codes to descriptive phrases.

An example of a report is:

event	count
function call	197393
function failure	441
function return	190679
function suspension	6272
function resumption	0
function suspension removal	6272

What does such a report suggest about the SP? The small number of failures indicates that most function calls are used for straightforward computation. The lack of resumptions of suspended function calls indicates that the few calls of functions that are generators are used just to get a single value.

Is this report typical of most programs? No, it isn't. In fact, function-evaluation profiles tend to vary widely, with some showing a large number of suspensions and resumptions.

What such profiles do show is the "character" of programs. The profile shown above can be characterized as showing "C-like" use of Icon, at least as far as functions are concerned. That is, it does not use functions as generators at all.

Next Time

What we've shown here is just a start. In subsequent articles on dynamic analysis, we'll show both more complicated kinds of MPs and lead up to the results of dynamic analysis on programs in the Icon program library.

References

1. "Evaluation Sandwiches", Icon Analyst 6, pp. 8-10.

2. "Memory Monitoring", Icon Analyst 1, pp. 7-10.

3. "Memory Monitoring", Icon Analyst 2, pp. 5-9.

4. "Multi-Thread Icon", Icon Analyst 14, pp. 8-12.

5. "Monitoring Icon Programs", Icon Analyst 15, pp. 6-10.

6. "Program Visualization", Icon Analyst 16, pp. 1-8.

What's Coming Up

We expect to have a second article on string invocation and another in the series of articles on dynamic analysis in the next issue of the Analyst.

In our occasional feature **From the Library**, we'll depart from our usual focus on the most useful material in the Icon Program library and present a drawing application that is more fun than useful.