The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1992
Number 10

In this issue …

Getting to the System

As a programming language matures, its computational repertoire tends to grow in response to user requests for more functionality. That's certainly true of Icon: Version 6 had 48 functions; Version 8 has 89.

The area of most demand for increased functionality is in access to operating-system facilities. Many such needs can be handled through the system() function.

The system() Function

The system() function in Icon (which just calls the corresponding C library routine) provides a gateway to the operating system command line. On most platforms, the argument of system() can be anything you'd type on a command line.

Since command-line facilities are necessarily operating-system dependent, the things you can do via system() and the syntax you use vary from system to system. For example, in UNIX, you can use

```
system("ls *.icn")
```

to get a listing of all files with the suffix .icn in the current working directory, while in MS-DOS, you'd use

```
system("dir .icn")
```

to get a similar result.

Using system() can be dangerous. You can do things like

```
system("rm *")
```

which deletes all files in the current working directory under UNIX.

It's not so much that you'd do this intentionally, but if the argument of system() is constructed in a complicated way, it may not come out as you expect. If the results are destructive, the consequences can be disastrous.

One trick we like to use when developing programs that use system() is to add the line

```
system := write
```

at the beginning of the program, which changes the value of system to the function write(). Then instead of actually calling the function system(), you get to see the argument with which it's called. And it covers all uses of system() with a single, easily removed line.

While system() lets you do many useful things from inside a running program, it has its limitations. The output of a command invoked by system() doesn't come back to your program. It goes to standard output or maybe standard error output unless you direct it to a file in the argument of system(), as in

```
system("ls *.icn >file.lst")
```

You can, of course, open and read a file to which you've directed the output of a command, but that's awkward.

Because of the way some operating systems work, system() may not always do what it would from the command line. For example, in UNIX,

```
system("chdir /usr/bin")
```

doesn't return with your current working directory changed to /usr/bin; the former current working directory is restored after executing in the shell for the system() command.

Another problem with system() is the value it returns, which is the exit code for the command it invokes. In theory, a nonzero exit code (on most operating systems) indicates some kind of error, while a zero exit code indicates successful completion of the command. For example, in UNIX, the diff command, which looks for differences in files, returns an exit code of 0 if the files are the same but an exit code of 1 otherwise. This might be used as

```
if system("diff " || file1 || " " || file2) = 0 then
   write(file1, " and ", file2, " are the same")
```

Not all programs return useful exit codes. For example, the UNIX program ls, which lists specified files, returns the exit code 0 whether or not there is any file that matches the specification (at least it does on the system we use).

Worse yet, exit codes are unreliable. It's up to the program you call via system() to return both an appropriate and reliable exit code. Not all do.

The system() function is not supported on all platforms on which Icon runs. It requires (at least) an operating system with a command-line interpreter. While most operating systems have a command-line interpreter, even if it is an optional alternative to a visual interface, not all do. The Macintosh is notable for the lack of a command-line interpreter. Even if there is a command-line interpreter, the system() function may not be provided in the C library for the compiler Icon uses. For example, the MPW sub-system for the Macintosh has a command-line interpreter, but MPW C does not support system().

Even if a C library supports system(), it may not always work properly. Most C compilers for MS-DOS include system() in their libraries, but there may be problems in using it. The problem in MS-DOS usually is the limited amount of memory available. If there's not enough room for COMMAND.COM, which system() uses, the system() function may fail silently or, worse, hang your computer.

On UNIX platforms, on the other hand, system() is reliable and generally works very well.

In any event, it's worth testing the functionality of system() on your platform before investing in its use. And, of course, system() is anything but portable.

Pipes

For operating systems that support pipes (such as UNIX and OS/2), there's even more you can do from within an Icon program. Not only can you invoke shell commands using pipes, but you can read or write to pipes.

(Note that the MS-DOS idea of pipes — writing the output of a program to a temporary file and then reading that file in another program — is bogus).

You open a pipe just like you open a file, but instead of giving the name of a file, you give a shell command in the same style as for system(). You also need to specify "p" as an open option.

For example, if you want to get the names of the files in the current working directory that have the suffix .icn, and you're running UNIX, the following will do:

```
names := open("ls ∗.icn", "rp")
```

The "r" is optional; a pipe (or file) is opened for reading unless you specify otherwise. The value of names is now an Icon file and every time you read from it, you can another line from the shell command

```
ls ∗.icn
```

(When the output of ls goes to a pipe, as it does here, the file names are given one per line.) For example,

```
namelist := [ ]
while push(namelist, read(names))
```

creates a list of all the files in the current working directory that have the suffix .icn.

Writing to a pipe is done in a similar fashion. For example,

```
sorter := open("sort", "pw")
```

opens a pipe to sort with the output going to standard output. Subsequently,

```
while write(sorter, process())
```

sorts the lines produced by process().

You might ask why it's worth going to all of the trouble to do this with a pipe, when you can just write to standard output and pipe that output through sort, as in

```
prog | sort
```

One advantage of piping the output through sort from inside the program itself is that sorting becomes part of the program — you don't have to remember to pipe the output of the program through sort every time you run the program.

A command opened as a pipe (or as the argument to system()) can contain pipes. For example, in UNIX

```
sorter := open("grep Section | sort", "pw")
```

sorts only those lines that contain the substring Section.

Don't overlook the fact that you can compile and run an Icon program from inside another one using either system() or a pipe. Here's a silly example:

```
run := open("icont –s – –x", "pw")
write(run,"procedure main();write(\"hello\");end")
close(run)
```

For a not-so-silly example of this kind of use of pipes, see interpe.icn in the Icon program library.

There are a few things you should know about pipes. A pipe cannot be opened for both reading and writing in Icon. Open pipes count toward the maximum number of files that can be open simultaneously. Furthermore, having more open pipes than your platform can support often leads to unpredictable failures in spawned processes. You should close a pipe when you're through using it. The result of closing a pipe is the exit code for the shell invoked.

If you're not accustomed to using pipes, it may take you a while to get used to the idea and perhaps a little longer to fully appreciate the possibilities. And if you're using Icon on a platform that doesn't support pipes, you may well become envious of those that do.

Procedural Encapsulation

In an earlier article on result sequences [1], we discussed the use of sequences as a programming tool — thinking in terms of sequences of values.

For this purpose, a sequence takes on a special meaning. It's almost like a value, although you can't treat it that way in Icon. You need, instead, to deal with expressions that can generate sequences of values.

An expression isn't a data object either, and the results it produces can be obtained only at the place in the program that it appears.

On the other hand, if you're programming in terms of sequences, you may want to use the same or similar sequences at different places in your program. One way to handle this is to duplicate the expression for a sequence at all the places it's needed. This not only is extra work, but it also has the usual problems with code duplication: it's error-prone, it's hard to maintain, and it increases program size.

One way of using an expression in more than one place in a program is to "capture" it with a co-expression, which *is* a data value. In some cases, this may be the best thing to do, but co-expressions are not generators and if you want to generate the values from a co-expression like you would from the expression itself, you need to use repeated alternation and worry about refreshing the co-expression for its next use. Another problem with co-expressions is the large amount of memory they require. And co-expressions are not supported for all implementations of Icon.

There is yet another method of capturing an expression as a data value: *procedural encapsulation*. This method is based on the observation that for a given expression *expr* and a procedure declaration

```
procedure p()
  suspend expr
end
```

the result sequences for *expr* and p() are the same.

For example, the expression

```
|(0 to 3)
```

generates the digits 0 through 3 endlessly: 0, 1, 2, 3, 0, 1, 2, 3, … If this expression is encapsulated in a procedure

```
procedure digits()
  suspend |(0 to 3)
end
```

then

```
digits()
```

also generates 0, 1, 2, 3, 0, 1, 2, 3, …

There are several reasons for encapsulating an expression in this way. One is that there is only a single instance of the expression, but its values can be generated in as many places in a program as they are needed, using only calls to the procedure.

Another advantage of procedural encapsulation is that an expression can be parameterized, as in

```
procedure digits(i, j)
  suspend |(i to j)
end
```

There's a third, less obvious, use for procedural encapsulation: Procedures, unlike expressions, are data values. Therefore you can assign a procedure to a variable, store it in a structure, pass it as an argument to another procedure, and so on. We'll have more to say about this in a future issue of the *Analyst*.

Before going on, we need to qualify our statement about the equivalence of the result sequences for an expression and a call of a procedure that encapsulates it. This statement is true only if the expression is *independent*, by which we mean that its result sequence does not depend on the time and place it is evaluated or on factors outside the expression itself. For example,

```
|(0 to 3)
```

is independent, but

```
|(0 to &line)
```

and

```
|read()
```

are not.

Procedural encapsulation can be extended in a number of ways. For example, expressions can be added before or after the suspension to record information, provide diagnostics, and so on. One model for this is

```
procedure p()
   prolog
   suspend expr
   epilog
end
```

All kinds of things are possible here. An example is

```
procedure p()
   static limit
   initial limit := 0
   limit +:= 1
   if limit > 10 then fail
   suspend expr
end
```

which shuts down the use of *expr* after 10 calls of p().

Don't forget that suspend has an optional do clause like the one for every. You don't see it used often, possibly because it wasn't added to Icon until Version 7. Here's an example of its use:

```
global p_count

procedure p()
   initial p_count := 0
   suspend expr do
      p_count +:= 1
end
```

Here p() has the same result sequence as *expr*, but a count of the total number of times p_count() is resumed is kept in a global variable.

If you want to do this for several procedures, you could keep the counts in a table that is subscripted by procedures:

```
global count
      :
count := table(0)
      :
procedure q()
suspend expr do
   count[q] +:= 1
end
```

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)

It may seem a little strange to subscript a table with a procedure, but procedures are values and a table can be subscripted by any kind of value.

Just to take this a little further, here's how you could write out the counts of procedure resumptions:

```
cntlist := sort(count, 3)
while write(image(get(cntlist)), " : ", get(cntlist))
```

Finally, don't forget "evaluation sandwiches" [2]. You might find a use for something like this:

```
procedure p()
   suspend 2(before, expr, after)
end
```

References

1. "Result Sequences", The Icon Analyst 7, August 1991, pp. 5-8.

2. "Evaluation Sandwiches", The Icon Analyst 6, June 1991, pp. 8-10.

The Anatomy of a Program — A Recognizer Generator

Several of our readers have suggested that we include detailed analyses of complete Icon programs as a regular feature of the Analyst. We've been reluctant to do this, since an adequate evaluation of even a fairly short program takes a lot of space. It also requires special knowledge to understand many programs; there aren't that many programs that are suitable for analysis in a newsletter like this one.

Another, less-obvious problem is that when we study a program carefully, we almost always can find ways to improve it.

Trying to describe a program intensifies this problem. It seems inappropriate to devote space in the Analyst to a program that can be made better, so we work on improving the program. But when we start to write about it again, we find more things to improve. Somehow, this process seems not to terminate.

We've finally bitten the bullet and are describing here a program that certainly is not perfect, but we're to the point where we don't see how to improve it and our guilty conscience for never quite getting this article done has taken the day.

There are many ways to approach the description of a program. The approach here is top-down, starting with a description of the problem, showing what's needed to solve it, proceeding to the organization of the program, and finally describing programming details.

The Problem

In the second edition of the Icon language book [1], there's a method for converting a context-free phrase-structure grammar into a recursive-descent recognizer for the corresponding language. Despite the problems with recursive-descent methods, such a recognizer actually can be useful, and it's only a short step to a parser, which has more uses. You'll find some examples in the Icon program library.

We'll use a slightly more general form of syntax description than that given in the book: a version of Backus-Naur Form in which nonterminal symbols are enclosed in angular brackets and alternatives are separated by vertical bars. If you're not familiar with BNF, see the classic reference [2] or one of the numerous books on formal languages and syntax description.

A simple grammar that illustrates this syntax-description notation is

```
<plist> ::= [ ] | [<args>]  | [<plist>]
<args> ::= , | ,<plist> | ,<args>
```

Each line contains a definition, or *production*, that defines a *nonterminal* symbol. <plist> and <args> are nonterminal symbols, ::= stands for "is defined to be", and vertical bars separate alternatives as mentioned above. All other symbols (here brackets and commas) are *terminal* symbols that stand for themselves.

The language defined by <plist> contains strings such as "[,[ ]]", "[,,,]", and "[[ ]]".

A recognizer for <plist>, produced by the method described in Reference 1, is shown in the box below.

```
procedure plist_()
   suspend {
      (="[ ]") | (="[" || args() || ="]") | (="[" || plist() || ="]")
      }
end

procedure args_()
   suspend {
      (=",") | (="," || plist()) | (="," || args())
      }
end

procedure main()
   while line := read() do {
      writes(image(line))
      if line ? (plist() & pos(0)) then
         write(": accepted")
      else write(": rejected")
      }
end
```

A Recognizer for <plist>

The Structure of a Production

The problem is a conceptually simple one: Analyze each production in the grammar and construct a corresponding recognizing procedure.

String scanning is the obvious tool to use for the analysis of the productions, but some care is needed to avoid a messy, bug-ridden program.

The analysis should be organized around the syntax of productions, which have the form

production → < *name* > ::= *rhs*

where *rhs* stands for "right-hand side". A right-hand side in turn is a sequence of alternatives separated by vertical bars:

rhs → *alt* | *alt* | … | *alt*

and an alternative is a sequence of symbols:

alt → *sym sym* … *sym*

The point here is that the hierarchical structure of a production should be reflected in the structure of its analysis.

Transforming a Production

The components of a production as described above also need to be transformed to produce a recognizing procedure. Following the method described in Reference 1, the transformation is:

```
τ(< name > ::= rhs) = procedure name()
                         suspend τ(rhs)
                      end

τ(alt | alt | … | alt) = τ(alt) | τ(alt) | … | τ(alt)

τ(sym sym … sym) = (τ(sym) || τ(sym) || … || τ(sym))

τ(terminal) = ="terminal"

τ(nonterminal) = nonterminal()
```

Organization of the Program

The hierarchical structure of a production is reflected in a hierarchy of procedures that analyze and transform a production:

transprod()	transform a production
↓	
transalts()	transform the alternatives
↓	
transseq()	transform a sequence of symbols
↓	
transsym()	transform a symbol

Writing the Program

There are two main issues in actually writing the program: (1) how to perform the analysis and (2) how to produce the required transformed output.

We've chosen a style of analysis that uses string scanning exclusively and in which "matching" procedures do the work. These matching procedures operate in the context of a scanning environment that is established before they are called. For example, the transformation loop in the main program could be written as

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

but we've chosen to write it instead as

```
while line := read() do
  line ? transprod()
```

The choice here is mostly a matter of personal preference. We can't say one way is better than the other. One reason we chose the second form is because it has a flavor of pattern matching that we think is interesting and is not used as often as it might be.

Thus, the procedure transprod() is called with the appropriate scanning environment already in place. transprod() needs to get the name of the nonterminal symbol being defined by the production, write a procedure heading, provide the "shell" for suspension, call tranalts() to transform the alternatives, and then finish off the procedure declaration:

```
procedure transprod()

  {
    ="<" &
    write("procedure ", tab(many(nchars)), "()") &
    =">::="
    } | error()

  write("   suspend {")
  transalts()
  write("      }")
  write("end")

  return

end
```

The global variable nchars contains a cset of the characters that are acceptable in a nonterminal name. Note that in writing the procedure heading, the second argument of write() is the result of string scanning, inserting the name in the proper place in the procedure declaration. Conjunction is used to bind the three expressions involved in analyzing the production into a single unit, so that only one call of error() is needed. In this program, an error in the syntax of a production terminates program execution. (It's hard to imagine how to recover from such an error in a useful way.)

What's next is the procedure transalts(). It is called with the same scanning environment as for transprod() but with the position now at the first character of the right-hand side.

Since a right-hand side is a sequence of alternatives separated by vertical bars, a loop is the natural control structure to use. Since vertical bars *separate* alternatives, an alternative is matched by

```
tab(upto('|') | 0)
```

This construction is generally useful when items are separated by some marker but there's no marker after the last one. It matches up to the separator or else matches the remainder of the subject.

If there were no transformation to apply to the subject, the analysis loop would look like this:

```
repeat {
  tab(upto('|') | 0) ? transseq()
  move(1) | break
  }
```

where

```
move(1) | break
```

either moves past the separator or terminates the loop in the case of the last alternative.

To construct the required output, however, parentheses are needed around each alternative and vertical bars are needed between them. The complete procedure, therefore, is a bit more complicated:

```
procedure transalts()

  writes("      ")
  repeat {
    writes(" (")
    tab(upto('|') | 0) ? transseq()
    if move(1) then writes(") |")
    else {
      write(")")
      return
      }
    }
end
```

Note that an alternation symbol is only written when there is another alternative to process.

Now on to transseq(). Since it's invoked as a matching procedure in

```
tab(upto('|') | 0) ? transseq()
```

the scanning environment for transseq() is just the current alternative. There is a question of how to terminate the loop: by checking in transseq() or having transsym() fail. To allow for the possibility of an empty alternative (that's perfectly reasonable), it turns out to be more convenient for transseq() to check when transsym() has processed the last symbol. Note that the symbols for concatenation are provided only when there is another symbol to process:

```
procedure transseq()

   repeat {
      transsym()
      if not pos(0) then writes(" || ")
      else return
      }

end
```

As noted above, there are two kinds of symbols and the translations are quite different in the two cases. Analysis and transformation of a nonterminal symbol involves a straightforward scanning expression with conjunction again used to bind the component sub-expressions. A terminal symbol, on the other hand, requires a prefix equals sign and enclosing quotes:

```
procedure transsym()

   if ="<" then {
      {
         writes(tab(many(nchars)), "()") &
         =">"
         } | error()
      }
   else writes("=", image(tab(upto('<') | 0)))

   return

end
```

The function image() is handy for producing the enclosing quotes — otherwise, escaped quotes in literal strings are required. These are hard to read and easy to get wrong. But image() does more. If a terminal symbol is a character that is significant syntactically in an Icon quoted literal, image() provides an escape sequence for it. Thus, a terminal symbol can be, for example, a quotation mark without causing any problems.

Note that if several terminal symbols occur in a row, they are combined into a single matching expression. For example, abcd gets transformed into

```
   ="abcd"
```

rather than

```
   ="a" || ="b" || ="c" || ="d"
```

as specified formally in the transformation given earlier. The result is the same and the first form is, of course, not only more compact but more efficient.

That's about it. There are a few more things that need to be taken care of in the complete program, such as writing out a main procedure that reads in lines and determines whether or not they are sentences in the grammar. It's also necessary to identify the "goal" nonterminal symbol, which we've chosen to be the one for the first production. The complete program is shown in the boxes on the next two pages.

Retrospective

If you look at the program, you may notice that there's not a single concatenation. Everything is arranged so that the output is written as the productions are analyzed. This is an example of the use of output to avoid concatenation within the program as described in Reference 3.

This didn't happen by accident. The program that appears here was originally written many years ago. The first version (which worked as well as this one) was a little horror. Its organization was confused, the use of string scanning was complicated and poorly structured, and most of the output for the recognizing procedures was built up using concatenation and only written out after the analysis of a production was complete.

Over the years, the program was incrementally improved and even completely rewritten several times. After the organization was made more logical, the use of string scanning was improved. Finally, we became interested in what amounted to a puzzle: "Could the program be written without any concatenation?" Considering the concomitant analysis and transformation, that clearly should be possible, and once we were convinced of that, it wasn't hard to do.

The evolution of this program parallels our personal development of skill and style in writing Icon programs. It's not that it took so long to arrive at the final (?) version of the program — we now can come much closer to producing a reasonable program on the first try.

We recommend refinement and rewriting as tools for learning to program well, not just for improving specific programs.

Finally, we have to admit that although we started out to write this article with a program we thought was in very good shape, we found many things to improve before this article was done. And now that the article is done, we've thought of more improvements and generalizations. But we have to stop somewhere.

Incidentally, just as refining and rewriting are good ways to learn to program better, writing about a program is an excellent way to find improvements. In having to explain every little aspect of a program, you are forced to focus on things you never really think about when you're just trying to get a program to work.

What Else?

What more could be done to the program here? Several things come to mind:

• Since the output of this program is another program, it might be useful to have a way to "pass through" Icon code from the input to the recognizer generator to its output. (If you can't think of a use for such a feature, we'll show one an upcoming issue of the Analyst.)

```
global goal                          # nonterminal goal name
global nchars                        # characters allowed in a  name

#
# Translate a grammar into a recognizer.
#

procedure main()
   local line                        # a line of input

   nchars := &letters ++ '_'

   while line := read() do {         # process lines of input
      line ? transprod()             # transform the production
      }                              # end while

   write("procedure main()")         # write out the main procedure
   write("   while line := read() do {")
   write("      writes(image(line))")
   write("      if line ? (", goal, "() & pos(0)) then ")
   write("         write(\": accepted\")")
   write("      else write(\": rejected\")")
   write("      }")
   write("end")

end

#
#  Transform a production.
#

procedure transprod()
   local sym                         # the symbol being defined

   {
      ="<" &                         # begin the procedure declaration
      write("procedure ", sym := tab(many(nchars)), "()") &
      =">::="                        # skip definition symbols
      } | error()                    # catch syntactic error
   write("   suspend {")             # begin the suspend expression
   transalts()                       # transform the alternatives
   write("      }")                  # end the suspend expression
   write("end")                      # end the procedure declaration
   write()                           # space between declarations
   /goal := sym                      # first symbol is goal

   return

end

#
#  Transform a sequence of alternatives.
#

procedure transalts()

   writes("      ")                  # write indentation
   repeat  {                         # process alternatives
      writes(" (")                   # open parenthesis for alternative
      tab(upto('|') | 0) ? transseq()    # transform the symbols
```

A Recognizer Generator (continued on next page)

• We've made a lot out of avoiding unnecessary concatenation. Here we've used the "output as a weak form of concatenation" ploy. Can you think of a way to avoid concatenation in the *recognizer*? *Hint:* It only requires changing a couple of characters in the recognizer generator.

• As shown in Reference 1, it's easy to convert a recursive-descent recognizer to a recursive-descent parser using lists to build a parse tree. That's a good exercise, but if you try it, you might think about using records for nonterminal types instead of using just lists.

• The main procedure provided for the recognizer here needs revising for a parser. While you're at it, provide a way for a person creating a parser to specify a main procedure.

• Using the present syntax for productions, the right-hand side metacharacters <, >, and | are excluded from the set of terminal symbols. Provide a way for specifying these characters as terminal symbols. *Hint*: One way to do this is to provide an escape mechanism. There's a more general and elegant approach. Look at rsg.icn in the Icon program library to see how this can be done in an entirely different way.

• There is no way of grouping symbols in the right-hand sides of productions. It would be useful if this could be done. For example,

```
<expr>::=<term><addop><expr>|<term>
```

could be written as

```
<expr>::=<term>(<addop><expr>|<empty>)
```

which not only is more compact, but would avoid backtracking and re-matching for <term> if it's not followed by an <addop>. Note that parentheses become metacharacters in this scheme.

Incidentally, we put in <empty> only for clarity. It's not necessary; the recognizer generator handles an empty alternative correctly. Thus, the production could be written as

```
   <expr>::=<term>(<addop><expr>|)
```

which looks peculiar but poses no processing problems.

And Around Again

Having posed these problems, we'll now have to solve them ourselves.

The result may be in the Icon program library before this issue of the Analyst is published.

References

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pages 180-186.

2. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," Backus, J., *Proceedings of the International Conference on Information Processing*, 1959, pp. 125-132.

3. "String Allocation", The Icon Analyst 9, pp. 4-7.

```
      if move(1) then writes(") |")      # if more, close the parentheses
                                         # and add the altrnation
      else {
        write(")")                       # no more,  close the parentheses
        return
        }                                # end else
      }                                  # end repeat
end

#
#  Transform a sequence of symbols.
#

procedure transseq()

   repeat {
      transsym()                         # process a symbol
      if not pos(0) then writes(" || ")  # if more, provide concatenation
      else return                        # else get out and return
      }                                  # end repeat
end

#
#  Transform a symbol.
#

procedure transsym()

   if ="<" then {                        # if it's a nonterminal
      {                                  # write it with suffix
        writes(tab(many(nchars)), "()") &
        =">"                             # get rid of closing bracket
        } | error()                      # or catch the error
      }                                  # end then
                                         # otherwise transform nonterminal
   else writes("=", image(tab(upto('<') | 0)))

   return

end

#
#  Issue error message and terminate execution.
#

procedure error()

   stop("*** malformed production: ", tab(0))

end
```

A Recognizer Generator (concluded)

Writing Bullet-Proof Programs

If you're using Icon to write an application for use by another person, you may need to give more attention than you otherwise might to making your program "bullet-proof" so that invalid data or a user error doesn't cause the application to crash.

If the application is critical, the importance of crash-resistance is obvious. Consider an application that is updating a data base and terminates with an error because there isn't enough memory to sort a table.

Even if the application is not critical in this sense, termination because of a run-time error can be very perplexing to a user who may know little or nothing about programming, much less what makes Icon tick.

The first requirement for writing a bullet-proof program is correctness. Easy to say. Despite the efforts of legions of software engineers, most programs contain errors, and all large, complex ones do.

But, as noted above, not all run-time errors result from programming errors. They may occur because of inadequate resources. Even if we pretend that Icon itself has no bugs, there are some aspects of its implementation that can cause problems. There are things you can do to reduce the chances of getting into trouble because of them.

From time to time, we'll discuss topics related to these matters. In this article, we start with two topics, one dealing with program correctness and another dealing with how to turn potential run-time errors into detectable expression failure.

Opening Files

One of the most common programming errors is not checking for failure when attempting to open a file. Misspelling is a common problem when the file name is provided by the user. Failure also may occur when opening a file for reading because it isn't there (although maybe it should be) or because the application lacks permission to read the file (possibly because of a mistake in assigning permissions). Failure also may occur when attempting to open a file for writing because the application lacks the necessary permission.

Failure in opening a file also can occur because you forgot to close files that were previously opened, leaving no more available file descriptors. This problem is common and insidious in its effects.

We recently had some personal experience with this. We have a on-line transaction system for processing orders for Icon material. It's written in Icon and does all kinds of things, including producing packing lists and mailing labels, as well as updating a data base that contains information about orders. This program had been working for over two years without a problem until one afternoon Robyn Austin, who handles Icon orders, had an unusually large number of orders to enter in one session. The program crashed, leaving things in disarray.

The cause of the problem was too many open files. Several files are opened and closed for each transaction, but one of them wasn't closed, so a file descriptor was consumed for each transaction. The fact that the program had been running for over two years without the problem arising is a reminder not to assume any program is free of bugs.

There are all kinds of other reasons why attempting to open a file can fail. A common mistake in application programs is the specification of a local path name instead of a full path name, so that the program runs properly for the developer in one area but fails when run in another area by the user.

Whatever the reason, if failure goes undetected when an attempt is made to open a file, all kinds of things can happen. If you're lucky, the program will terminate quickly with an error. If you're not, you may think you're writing data when you're not. Consider the following program segment:

```
output := open(filename, "w")

while write(output, read(input))
```

If open() fails, no value is assigned to output. Unless something has been assigned to output previously, it has the null value. The null value as the first argument to write() is taken as an empty string (not a file). So data from standard input is written to standard output instead of to the intended file. A user may be annoyed as well as baffled.

Something slightly different happens when failure occurs on opening a file for input. Consider

```
input := open(filename, "r")

while write(output, read(input))
```

If the value of input is null before open() is called, as it is likely to be, no assignment is made to input if open() fails, and the value of input still is null. A null argument to read() defaults to standard input. Instead of reading from the expected file, read() waits for standard input. If standard input comes from the keyboard, the program appears to hang when, in fact, it is just waiting. A panicked user who thinks the program is in a loop may interrupt execution of the program, perhaps at a very unfortunate time. *Advice*: If you think your program is in a loop, try entering an end of file from the keyboard before interrupting the program.

It's easy enough to check for failure in open():

```
output := open(filename, "w") |
  stop("*** cannot open " , filename)
```

An abstraction such as Open(name, attrib), which does the same thing as open(name, attrib), but terminates with an error if opening the file fails, is a useful way of avoiding these problems without having to write a lot of code every place a file is opened. A procedure to do this is:

```
procedure Open(filename, attrib)

  return open(filename, attrib) |
    stop("*** cannot open ", filename)

end
```

Error Conversion

It may be reasonable to terminate with an error message if a file cannot be opened, but it may be disastrous to let a program terminate if sorting a table fails because of inadequate memory as mentioned at the beginning of this article. In such an critical situation, error conversion may be the solution.

Error conversion, which converts a run-time error to expression failure, is a crude and broad-brush tool. It applies to all run-time errors (or almost all; see the next section). You can't specify that you want only certain kinds of errors to be converted to expression failure.

The indiscriminate nature of error conversion sometimes causes problems. It can cause failure at unexpected places. Such failure may go undetected or introduce ambiguous failure, both of which are well-known causes of programming errors in Icon.

For example, if error conversion is in effect,

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

may not do what it seems to do. The loop may end because of an end of file on input, which is what you'd expect. But the loop also may terminate because of an error in reading or writing. A reading error could be bad media or a hardware problem — less likely these days than it used to be, but still possible, especially on personal computers. A writing error also can occur because of lack of file space, especially when output is directed to a floppy disk.

Since there's nothing to distinguish between these cases, a loop that terminates because of an error may give the erroneous appearance of a successful file copy.

In the situation above, a simple solution is

```
&error := 1

while write(read())

if &error = 0 then stop("Error occurred in copying")
```

A slightly more sophisticated approach is to use an error-checking procedure that allows the user to decide what to do:

```
procedure ErrorCheck(line, file, message)

   write("Error occurred ", message)
   write("   error: ", &errornumber)
   write("   line: ", line)
   write("   file : ", file)
   write("   cause: ", &errortext)
   write("   offending value: ", image(&errorvalue))
   writes("Do you want to continue? (n) ")
   if map(read() == "y" | "yes") then return
   else exit(&errornumber)

end
```

Then the loop and check above could be written

```
&error := 1

while write(read())

if &error = 0 then
   ErrorCheck(&line, &file, "in copying")
```

The Limitations of Error Conversion

Although most run-time errors, including inadequate storage to create an object, can be converted to failure, there are a few that cannot:

• Start-up errors — problems that occur before program execution actually begins.

• Errors that are trapped by the system on which Icon runs. Division by zero is a typical example. The reason such errors cannot be converted to failure is that the implementation of Icon loses contact of where it was when the error occurred.

• Errors for which conversion to failure would have an inordinate impact on the implementation and performance of Icon. There are a few of these, but you're not likely to run into them in practice. One is inadequate space for converting a large integer to a string. Another occurs in dereferencing if a subscripted string-valued variable has changed to another type. An example is:

```
x := "abc"
x[2] := ((x := [  ]) & "B")
```

Such a construction is rather unlikely — and a programming error, in any event.

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

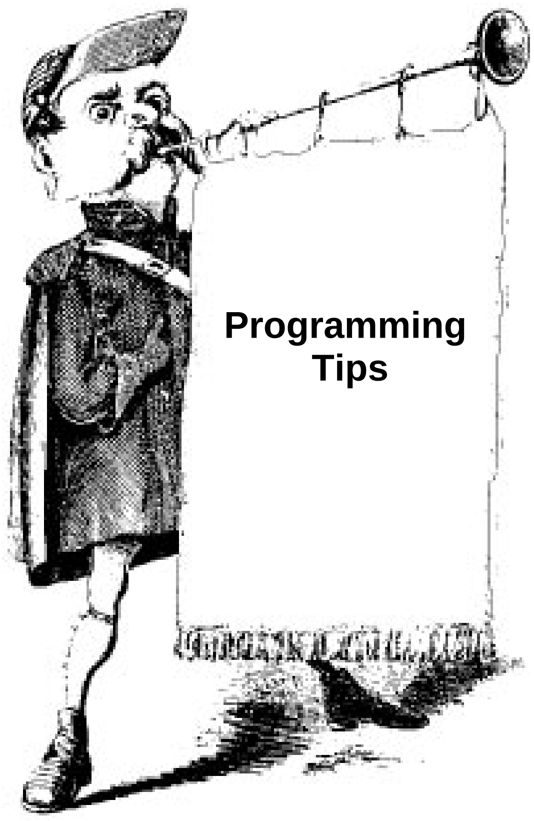
or

...{uunet,allegra,noao}!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona



The Generality of String Scanning

The subject of a scanning expression is a string, usually provided as the value of an identifier, as in

```
while line := read() do
  line ? {
    if ="Section " then {
      name := tab(0)
      break
      }
    }
```

which skips to a line in standard input that starts with the string "Section ".

It’s worth remembering, however, that the subject of string scanning doesn’t have to be such a simple expression — it can be any expression, even a generator. For example,

```
(line1 | line2 | line3) ? find(token1 | token2)
```

succeeds if one of line1, line2, or line3 contains either token1 or token2. Since the analysis expression tries all possibilities on the current subject, this scanning expression only goes on to line2 if neither token1 nor token2 is contained in line1, and so on.

A generator as the subject is particularly useful for examining all the values in a structure. For example, if words is a list of strings,

```
!words ? upto(&ucase)
```

succeeds if any of the strings in words contains an uppercase letter. Similarly,

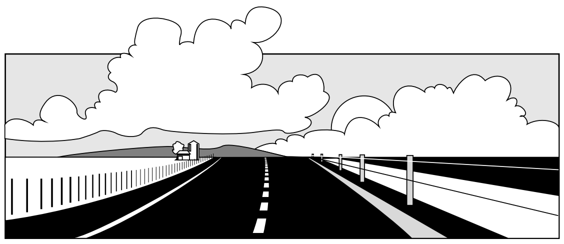
```
every write(
  !words ? {
    upto(&ucase) &
      tab(0)
    }
  )
```

writes all the words containing an uppercase letter. Note that upto() itself does not change the position in the subject, so tab(0) matches the entire subject. Constructions like these work for sets and records as well.

With these possibilities in mind, the loop at the beginning of this article can be recast more compactly as

```
!&input ? {
  ="Section " &
    name := tab(0)
  }
```

This is a case where the element-generation operation applied to a file works very naturally. A line from standard input is generated and the analysis expression is applied to it. If the analysis expression succeeds, the entire expression succeeds and no more lines are read. If the analysis expression fails, however, the element-generator operation is resumed to produce another line from the input file, and so on. Note that if there isn’t a line that begins with "Section ", the entire input file is consumed. A test should be added to determine if the entire expression succeeds or fails.



What’s Coming Up

We’ve made a point from time to time about the importance of choosing good representations for data that is manipulated by Icon programs. In the next issue of the Analyst, we’ll present a case study showing the consequences of different data representations. The results may surprise you.

We’ll also have an article on modeling Icon functions using procedures — a process that can help you learn more about exactly what Icon functions do.

The next issue also has an article on command-line arguments; how they can be used to pass data into a program and to specify processing options.