The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1991
Number 5

In this issue …

String Scanning Examples

Examples seem to be particularly helpful to persons who are learning to use string scanning. This article presents several such examples. Some of the examples are comparatively simple; if you are an experienced Icon programmer, you may wish to skip the examples that you know you can handle. Some suggested exercises are provided in case you'd like to test your string-scanning proficiency.

Simple Transformations

Many string-processing tasks can be cast as string transformations in which an input string is scanned while an output string is built up — the iterative string scanning paradigm mentioned in Issue 4 of the Analyst. Such string transformations often are best cast as procedures that can be used in a variety of situations.

Removing all instances of specified characters in a string provides a simple example:

```
procedure remove(instring,c)
  local outstring

  outstring := ""
  instring ? {
    while outstring ||:= tab(upto(c)) do
      move(1)
    outstring ||:= tab(0)        # don't forget the rest
    }
  return outstring
end
```

In this model of string transformation, the string being produced starts out empty and is built up by appending the appropriate substrings of the string being scanned, skipping characters is the cset c. The names instring and outstring are used to emphasize the relationship between the string being scanned and the result being produced. In practice, other names may be more appropriate.

This procedure can be improved by skipping over consecutive characters in c:

```
procedure remove(instring,c)
  local outstring

  outstring := ""
  instring ? {
    while outstring ||:= tab(upto(c)) do
      tab(many(c))
    outstring ||:= tab(0)        # don't forget the rest
    }
  return outstring
end
```

In early versions of Icon, returning from inside a scanning expression did not restore the scanning environment outside it — and hence could play havoc with a scanning expression unrelated to the procedure. Consequently, returning from inside a scanning expression was a bad programming practice. Starting with Version 7 of Icon, this problem was fixed, so that a return from inside a scanning expression does not cause trouble elsewhere. Many Icon programmers still follow the old rule, but it's worth noting that such procedures can be written more compactly by returning at the point where the return is natural:

```
procedure remove(instring,c)
  local outstring

  instring ? {
    outstring := ""
    while outstring ||:= tab(upto(c)) do
      tab(many(c))
    return outstring || tab(0)
    }
end
```

We've put the initialization of outstring inside the scanning expression to emphasize that it is part of the transformation process. That's just cosmetic; its scope is the same either way.

Compressing runs of characters in a cset provides another example of the transformation paradigm:

```
procedure compress(instring,c)
  local outstring, c1

  instring ? {
    outstring := ""
    while outstring ||:= tab(upto(c)) do {
      outstring ||:= (c1 := move(1))
      tab(many(c1))
      }
    return outstring ||:= tab(0)
    }
end
```

Exercise: Write a procedure to delete all occurrences of one string that occur in another.

These procedures take a "positive" view of scanning — they look for the specified characters. Here's a "negative" approach, taken from a program submitted to the Icon program library with only the variable names changed to corresponds to those used in previous examples here:

```
procedure compress(instring,c)
  local outstring, c1

  instring ? {
    outstring := ""
    while outstring ||:=  tab(many(~c)) do {
      outstring ||:= (c1 := move(1))
      tab(many(c1))
      }
    return outstring || tab(0)
    }
end
```

This procedure is incorrect in several respects. In the first place, instring may start with a character that is in c. In this case, the loop fails immediately, and the procedure produces the string unchanged. If the string does not contain any characters in c, move(1) fails, c1 has the null value, and there's a run-time error. Finally, if the string ends in several different characters that are in c, only the first one is compressed. It may seem unlikely that mistakes as obvious as these would go unnoticed, but there are some situations in which they might not be detected for some time. For example, if the procedure is used to compress runs of blanks between words, it may work well for much typical data — lines of natural-language text usually do not begin or end with blanks or lack blanks altogether.

When we decided to show this example, we thought we'd fix it up so that it could be compared to the "positive" approach. That's not so easy to do — you might want to try it.

In any event, this kind of programming technique seems awkward to us, and we've seen enough other examples of incorrect scanning expressions using the "negative" approach that we advocate the "positive" one.

It's also worth noting that a cset construction operation like complementation should not be used in a loop, since Icon builds a new cset each time, which is expensive in both time and storage throughput. If you really want to use the "negative" approach, build the complemented cset once, before starting scanning.

Columnar Data

The data in a spreadsheet is organized in rows and columns. Each row is a "record" and each column is a "field" of a record. Most applications that deal with such data store it in a proprietary format that suits their needs. Nearly all such applications, however, can "export" such data as pure text and similarly "import" pure text, provided it is formatted properly. In one common text format, each record is a line consisting of fields separated by tabs.

Suppose you want to reformat such tab-separated data so that it can be printed with the fields aligned in columns. For sake of example, suppose the columns are 10 characters wide with the data left-aligned.

We'll start with a scanning expression that transforms tab-separated data in an identifier instring into fixed-width fields in an identifier outstring:

```
instring ? {
  outstring := ""
  while field := tab(upto('\t') | 0) do {
    outstring ||:= left(field,10)
    move(1) | break
    }
  }
```

Since instring contains tab-*separated* fields, not tab-*terminated* fields, there is no tab after the last field. This is handled by tab(upto('\t') | 0). That is, the cursor is moved up to the next tab, if there is one, but to the end of the subject if there isn't. The move(1) in the do clause moves past the tab, if there is one. It fails and the loop ends, via break, if there isn't a tab.

The auxiliary identifier field can be eliminated by appending the fixed-width fields directly in the control clause of the while loop:

```
instring ? {
  outstring := ""
  while outstring ||:= left(tab(upto('\t') | 0),10) do
    move(1) | break
  }
```

When you first start writing scanning expressions, it's probably a good idea to keep the analysis — tab(upto()) — separate from the synthesis — outstring ||:= left(). After things are working, you can go back and make refinements, such as combining the analysis and synthesis. As a general rule, taking the trouble to rethink scanning expressions and to make refinements is worth the effort. It will enable you to write better scanning expressions in the long run.

The way a string scanning expression is structured depends to some extent on how it will be used. In the example above, the desired value is produced as a side effect of string scanning. You might prefer to have the scanning expression *produce* the value. This can be done by making outstring the last component of the analysis expression:

```
instring ? {
   outstring := ""
   while outstring ||:= left(tab(upto('\t') | 0),10) do
      move(1) | break
   outstring
   }
```

In this case, the result of the scanning expression is outstring, since it's the last expression in a compound expression. That is, because of automatic semicolon insertion, the expression above is the same as:

```
instring ? {
   outstring := "";
   while outstring ||:= left(tab(upto('\t') | 0),10) do
      move(1) | break;
   outstring;
   }
```

You might use this formulation as follows:

```
while instring := read() do
   write(
      instring ? {
         outstring := ""
         while outstring ||:=
            left(tab(upto('\t') | 0),10) do
               move(1) | break
         outstring
         }
      )
```

or even as :

```
while write(
   read() ? {
      outstring := ""
      while outstring ||:=
         left(tab(upto('\t') | 0),10) do
            move(1) | break
      outstring
      }
   )
```

The trouble with getting the desired result from string scanning by making an identifier the last component in the analysis expression is that it may not be easy to see at a glance what's going on.

Another common text format for spreadsheet data is comma-separated, in which each field is enclosed in quotes (to protect commas that may occur within fields). It's easy enough to adapt the scanning expressions above to this input format.

Exercise: Adapt the scanning expressions above to handle comma-separated data.

A slightly more interesting problem is the conversion of tab-separated data into comma-separated data. For simplicity, we'll use the first format for scanning as given above, so that the desired result is produced as a side effect:

```
instring ? {
   outstring := ""
   while field := tab(upto('\t') | 0) do {
      outstring ||:= image(field) || ","
      move(1) | {
         outstring ?:= tab(–1)
         break
         }
      }
   }
```

This formulation uses image(), as suggested in Issue 3 of the *Analyst*, to provide the quotes around the fields. Note the use of augmented string scanning to remove the trailing comma. As before, the analysis and synthesis operations can be combined:

```
instring ? {
   outstring := ""
  while outstring ||:= image(tab(upto('\t') | 0)) || "," do
      move(1) | {
         outstring ?:= tab(–1)
         break
         }
   }
```

The slightly awkward aspect of these formulations is that while the fields are supposed to be comma-*separated*, the synthesized string winds up with an unwanted comma after the last field. This comma can't be ignored, since it corresponds to an empty field in comma-separated data. Hence the truncation of outstring before leaving the while loop.

An alternative formulation is:

```
instring ? {
   outstring := ""
   while field := tab(upto('\t')) do {
      move(1)
      outstring ||:= image(field) || ","
      }
   outstring ||:= image(tab(0))
   }
```

We prefer this formulation to the former one, since it's easier to understand, but the choice between formulations is mostly a matter of taste.

We've deliberately ignored the possibility that quotation marks might occur within fields. It's clear what image() does, but that's not what most spreadsheet applications do (they vary somewhat, so a solution to this problems depends on what a particular application expects).

Exercise: Adapt the scanning expression above to convert double quotes in fields to single quotes.

One other format conversion is worth considering: converting fixed-field data to tab-separated data. Where all fields have the same width, it's easy. If the fields are, say, 10 characters wide and blank-filled on the right, then here's that's needed:

```
instring ? {
   outstring := ""
   while field := move(10) do
      outstring ||:= trim(field) || "\t"
   outstring ?:= tab(–1)
   }
```

Exercise: Combine the analysis and synthesis portions of this scanning expression and also provide a way to avoid adding a final tab that has to be removed.

The problem is a little tricker if the data is aligned at the right so that the padding blanks are at the left. One solution is:

```
instring ? {
   outstring := ""
   while field := move(10) do
      outstring ||:= reverse(trim(reverse(field))) || "\t"
   outstring ?:= tab(–1)
   }
```

Another solution is:

```
instring ? {
   outstring := ""
   while field := move(10) do
      outstring ||:= {
         field ? {
            tab(many(' '))
            tab(0) || "\t"
            }
         }
   outstring ?:= tab(–1)
   }
```

While this may seem to be a painful way to remove initial blanks, it's worth remembering that scanning expressions can be nested; something like this may be needed for situations in which there aren't functions in the built-in repertoire like reverse() and trim() to do the job.

Sometimes fixed-field data that needs to be converted to a tab-separated format has fields of different widths. A common case occurs in address records, where each field has a different width. For example, such records may have a 40-character name field, an 80-character address field, and a 10-character zip-code field. Assuming the data is left-aligned, converting such records to tab-separated form is a snap:

```
outstring := {
   instring ? {
      trim(move(40)) || "\t" || trim(move(80)) || "\t" ||
         trim(move(10))
      }
   }
```

It's worth noting that this scanning expression fails if the value of instring is less than 130 characters long.

Exercise: Modify this scanning expression so that it fails if the value of instring is *more* that 130 characters long.

Alternatively, you may just want to take the record apart and assign its fields to specific identifiers:

```
instring ? {
   name := trim(move(40)) &
   address := trim(move(80)) &
   zipcode := trim(move(10))
   }
```

The conjunctions deserve note. They bind the three components of the analysis expression together, so that if any one fails, the entire scanning expression fails.

Exercise: Modify this scanning expression so that it fails if the value of instring is more than 130 characters long.

If you know that the value of instring has the right length, an alternative formulation is:

```
instring ? {
   name := trim(move(40))
   address := trim(move(80))
   zipcode := trim(move(10))
   }
```

Scanning Backwards

Although tab(i) and move(i) can move the cursor to the left as well as to the right, there's a strong bias toward left-to-right scanning. Most data is organized this way, reflecting both the characteristics of the majority of natural languages and the way data is stored in computers. There are, however, times when scanning from right to left is useful.

One right-to-left situation occurs when you're building up a string based on the subject, but you want portions from the right end first.

Consider a procedure rotate(instring,i) that produces the result of rotating the value of instring left by i characters. This transformation amounts to dividing instring into two pieces and concatenating them in reverse order. One method is:

```
procedure rotate(instring,i)
  local first

  if instring == "" then return ""
  i %:= *instring

  instring ? {
    first := tab(i + 1)
    return tab(0) || first
    }
end
```

To begin with, i is reduced modulo the length of instring. Then the first part is obtained by scanning to the right and is appended to the remainder of the subject. There's nothing wrong with this approach, but the auxiliary identifier first can be avoided by moving to the right end of the string and scanning to the left:

```
procedure rotate(instring,i)
  if instring == "" then return ""
  i %:= *instring

  instring ? {
    tab(0)            # get to right end
    return tab(i + 1) || tab(1)
    }
end
```

Exercise: Modify these procedures to handle negative values of i as rotation to the right.

Sometimes more complicated kinds of analyses also are better done from right to left. Consider inserting commas in an integer in order to separate groups of three digits to produce the conventional written form. Thus, for example, "1234567" is transformed into "1,234,567".

There are lots of ways to perform this transformation, including mapping (see numbers.icn in Version 8 of the Icon programming library). To use string scanning, it's fairly evident that the comma insertion needs to be done from right to left. As in many such cases, there's a choice between reversing the string and working left to right or taking the string as-is and working from right to left. We'll start with reversal, since it at least transforms the problem into a familiar domain:

```
procedure commas(instring)
  local outstring

  outstring := ""
  instring := reverse(instring)
  instring ? {
    while outstring ||:= move(3) || ","
    if pos(0) then outstring ?:= tab(–1)
    else outstring ||:= tab(0)
    }
  return reverse(outstring)
end
```

The sticky point is that if the length of instring is an even multiple of three, the last comma needs to be removed.

Exercise: Rewrite this procedure so that a comma is not appended if the length of instring is an even multiple of three.

This procedure can be restructured to combine some of the operations:

```
procedure commas(instring)
  local outstring

  outstring := ""
  return reverse(
    reverse(instring) ? {
      while outstring ||:= move(3) || ","
      if pos(0) then outstring ? tab(–1)
      else outstring || tab(0)
      }
    )
end
```

Scanning reverse(instring) instead of first changing instring to its reversal is obvious, once you think about it; it's a reminder that the subject expression is not limited to being a string-valued variable. Returning the reversal of the result of scanning also is obviously what's needed. The only tricky thing here is that the result of scanning is outstring: the result of an if-then-else expression is the result of the expression in the selected clause. Admittedly, this is a bit obscure, and the economy obtained may not be worth the loss in program readability.

But what about just scanning from right to left and avoiding all this reversal? Here's a way:

```
procedure commas(instring)
  local outstring

  outstring := ""
  return instring ? {
    tab(0)                    #get to right end
    while outstring := "," || move(–3) || outstring
    if pos(1) then outstring ? {
      move(1)
      tab(0)
      }
    else tab(1) || outstring
    }
end
```

Here's a case where you might want to use

```
outstring[2:0]
```

in place of

```
outstring ? {
  move(1)
  tab(0)
  }
```

Whether you chose reversal or right-to-left scanning probably depends on what seems easiest. You might wonder if there's a difference in efficiency in the two approaches. Not much, really. The main difference is that the reversal method allocates more string storage.

Exercise: Modify the procedures above to handle signed integers.

Style

Despite all our remarks about good and bad practice in formulating string scanning expressions, string scanning is largely a matter of style. Different programmers may write entirely different kinds of scanning expressions to accomplish the same results.

You may not like our style. The fact is that we may not like the style we used last month. We just suggest that you develop guidelines for writing scanning expressions and try to use them in a consistent manner.

More to Come

We'll continue to feature articles on string scanning. In addition to the one on pattern matching in this issue of the Analyst, in future issues we'll explore some issues like the use of generators in the subject and analysis expressions and working with structures during string scanning.

Pattern Matching

Note: This article is based on material in the second edition of *The Icon Programming Language*. The approach here is somewhat different and the treatment is more detailed.

Pattern matching, as we use the term, refers to a view of string scanning in which expressions are thought of in terms of the strings they can match instead of how they do it. For example, move(i) can be thought of in two ways: (1) as an expression that adds i to the current position and returns the substring between the previous and new positions, or (2) as a pattern that matches any string that is i characters long. We'll concentrate here on the second view.

Although pattern matching is primarily a matter of viewpoint, it provides a powerful conceptual tool for analyzing strings.

The simplest patterns are what we've called *matching expressions*. There are two built-in matching expressions, move(i) and tab(i). More complicated matching expressions can be built by combining the built-in ones and by writing *matching procedures*.

The Matching Protocol

But there's more underlying this. What *is* a matching expression? A matching expression is defined in terms of a protocol that specifies what is allowed:

- A matching expression may increase the position (or leave it where it is).
- When a matching expression sets a new position, it suspends, producing the substring of the subject between the previous and new positions.
- If a matching expression is resumed and has no other position to set, it restores the position to its original value and fails.
- A matching expression may not change the subject.

The requirement that the position can only be increased is not really essential to the concept of pattern matching, but it simplifies some of the discussion that follows. It also is natural for most string analysis in Icon and fits nicely with string analysis functions, which always produce positions at the current position or to its right. Note that tab(i) and move(i) may violate this requirement. We'll assume they are not used that way in the discussion that follows.

There are two fundamentally different aspects to these rules of protocol: what a matching expression produces (the matched portion of the subject) and data backtracking (maintenance of the position).

The idea behind producing the matched portion of the subject is that a matching expression, viewed as a pattern, produces a specific string from among all those that the pattern can match. Thus, while move(i), as a pattern, characterizes all strings of length i, evaluating move(i) at a particular position in a particular subject produces a specific string of length i.

Since matching expressions may advance the position when they match, they have the nice property that the string matched by two matching expressions in succession is the concatenation of the strings they match. For example,

```
move(i) || move(j)
```

is a matching expression.

Data backtracking is more subtle. It means that a matching expression is responsible for maintaining the position and, in particular, leaving it as it was if the matching expression fails. This assures that alternative patterns are applied at the same place in the subject, and it corresponds to the intuitive notion of matching one pattern or another. For example, in

```
(move(5) || ="#") | tab(0)
```

if move(5) succeeds but ="#" fails, tab(0) matches starting at the same place as move(5) did.

In order for a matching expression to restore the position, it must suspend so that it can regain control if a subsequent matching expression fails. In the example above, move(5) suspends and is resumed when ="#" fails. The suspension has nothing to do with producing other matches. Although some matching expression are generators, move(5) can match in only one way. It suspends only so that it can restore the position if a subsequent matching expression fails. Of course, a matching expression that does not change the position need not suspend (but it does need to return the empty string).

Composing Matching Expressions

The ability to build up complicated matching expressions from simpler ones is essential to the use of pattern matching. The rules of protocol given above determine how matching expressions can be combined to form other matching expressions. The two basic forms of combination are concatenation and alternation. If *expr1* and *expr2* are matching expressions, then

expr1 || *expr2*

and

expr1 | *expr2*

are matching expressions. It's also the case that if *expr* is a matching expression,

|*expr*

is a matching expression, although it's usually not very interesting — consider what |move(1) does. And, of course, =s is a matching expression, since it's just a shorthand for tab(match(s)).

Note that, in general,

expr1 & *expr2*

is not a matching expression, since it produces the string matched by *expr2*, not the concatenation of the strings matched by *expr1* and *expr2*. Conjunction does, however, handle data backtracking properly. We'll come back to this point in a subsequent article, since there are cases in which it's useful to know if a pattern matches but it's not necessary to know what it matches. This is called *recognition* instead of matching and can be done with a simpler protocol than the one needed for matching.

Most operations on matching expressions do not yield matching expressions. Some don't because they do not produce the correct result, as in

expr1 + *expr2*

But you'd hardly expect to add two matching expressions. The more serious limitations arise with control structures.

By definition, control structures alter the flow of control. Most control structures have control expressions that determine what expression is evaluated next. Control expressions are *bounded*, so that if they produce a result, they cannot be resumed. Bounded expressions are fatal to matching because they prevent the backtracking that is required to maintain the position. For example, if *expr1* and *expr2* are matching expressions,

if *expr1* then *expr2*

is not, in general, a matching expression, because *expr1* is bounded and cannot be resumed to restore the position if a subsequent matching expression fails.

In general, a matching expression cannot appear in a bounded expression and maintain the required protocol. That covers a lot of ground. For example, in

{*expr1*; *expr2*}

expr1 is bounded. Similarly, in

expr \ i

expr cannot be resumed if it produces i results. The exclusion of bounded expressions applies only to matching expressions. For example, if *expr1* and *expr2* are matching expressions, then so is

if pos(i) then *expr1* else *expr2*

since the expressions in the then and else clauses are not bounded and pos(i) does not change the position (or contribute the result produced by the if-then-else expression).

Despite possibilities like this, matching expressions usually are composed by the concatenation and alternation of other matching expressions. These two operations correspond to "match this then match that" and "match this or match that", respectively.

Although the built-in repertoire of matching expressions is meager, the analysis functions that provide arguments for them allow the construction of fairly complex patterns. For example,

```
tab(find(":=" | "<–") + 2) ||
  (tab(many(' ')) | move(0)) ||
    tab(many(&letters))
```

is a pattern that matches any string that contains the substring ":=" or "<–" followed by a string of letters with possible intervening blanks. The pattern matching flavor of this expression is exemplified by

```
tab(many(' ')) | move(0)
```

to match an optional string of blanks. This also could be written as

```
move(0) | tab(many(' '))
```

The result is the same; the choice depends on whether blanks are likely. Similarly, "" could be used in place of move(0) to match an empty string.

Matching Procedures

In spite of what can be done in pattern matching with the built-in repertoire, you'll immediately think of reasonable patterns that require more capabilities. An example is a pattern that matches two simpler patterns in succession, but with a “gap” of intervening characters. (Well, you *can* write this with the built-in repertoire, but not in a reasonable way.)

The obvious approach to extending the built-in repertoire of matching expressions is to write *matching procedures* — procedures that obey the matching protocol.

Let’s start by seeing how you could write move(i) and tab(i) as procedures if they weren’t in the built-in repertoire. Figuring out what these procedures need to return is easy and we’ve emphasized suspension and restoration of the position sufficiently so that those aspects of the protocol should follow naturally. Here’s a start for tab(i):

```
procedure tab(i)
  local saved_position

  saved_position := &pos
  suspend .&subject[.&pos : &pos := i]
  &pos := saved_position
  fail

end
```

The local identifier saved_position is used to save the position. The argument of the suspend expression is the matched substring of the subject. There are two points to be noted here, one subtler than the other. The variable &pos in the first argument of the subscripting expression is dereferenced so that the assignment to &pos in the second argument does not change the first argument before it’s used. That’s just a matter of remembering that Icon does not dereference variables until all arguments for an operation are evaluated. The reason for dereferencing &subject is less obvious. When a procedure produces a result that is a variable, the variable is not dereferenced unless it’s local to the procedure. &subject[…] is a variable that is not local to the procedure, so it is not automatically dereferenced. If the dereferencing were not forced as it is above, it would be possible to assign to the result of tab(i) and consequently change the value of &subject. That may sound intriguing, but it’s not the way the built-in version of tab(i) works. Of course, if you forget to dereference the result produced by the procedure, you might go a long time without noticing this problem.

Back to more important things. Note that in the procedure above, if i is out of range, assignment to &pos fails, the argument of suspend fails, and the procedure does not produce a result. Assigning the value of saved_position to &pos changes nothing, since both have the same value in this case.

If i is in range, however, the position is changed and tab(i) suspends with the matched substring. If tab(i) is resumed, its remaining task is to restore the position and then fail. The fail expression is unnecessary, since flowing off the end of the procedure body accomplishes the same thing.

The procedure above can be written more concisely using Icon’s reversible assignment operation to take care of data backtracking:

```
procedure tab(i)
  suspend .&subject[.&pos : &pos <– i]
end
```

Reversible assignment suspends so that it can perform data backtracking case a subsequent expression fails. Next the proecdure call tab(i) suspends. If tab(i) is resumed, the suspended reversible assignment operation is resumed. It restores the position and then fails. (Note the similarity of reversible assignment and tab(i).) With this formulation, there is no need for a local identifier to keep track of the position and restore it. Finally, using the common Icon practice, failure is provided implicitly by flowing off the end of the procedure.

This procedure is an instance of a more general model for matching procedures:

```
procedure …
  suspend .&subject[.&pos : &pos <– new position]
end
```

where *new position* indicates an expression that produces one or more new positions for the position. An example of the use of this model is:

```
procedure gap()
  suspend .&subject[.&pos : &pos <–
    &pos to ∗&subject + 1]
end
```

This procedure matches strings of length 0, 1, … through the end of the subject.

As mentioned earlier, a pattern abstractly characterizes a set of strings without any particular order among them. A procedure like gap() may match several different strings, but, of course, there’s an order in how it goes about this. The procedure above is “pessimistic” and tries the shortest possible string first, matching longer strings only if it’s forced to do so by resumption resulting from the failure of subsequent matching expressions. An optimistic version of this matching procedure is:

```
procedure gap()
  suspend .&subject[.&pos : &pos <–
    *&subject + 1 to &pos by –1]
end
```

So far we’ve not used one of the potentially most powerful aspects of matching procedures — that they are first-class values.

To see why this might be useful, consider some matching procedure, p(). Suppose you encounter a situation in which you want to either match p() or the empty string. If you just need to do it once, you can use

```
p() | ""
```

What if you find you need this construction in a lot of places? You could make a modified version of p(). But suppose you also find that you want to use several different procedures this way.

The solution is to encapsulate the concept — match something or the empty string — in another procedure, say orempty(p). The idea is that orempty(p) applies p in the desired context:

```
procedure orempty(p)
  suspend (p() | "")
end
```

The parentheses around the argument to suspend are not necessary. Since this type of construction is used frequently, its worth remembering that such a procedure can be written as:

```
procedure orempty(p)
  suspend p() | ""
end
```

You may want to convince yourself that orempty() really does what we claim. Follow through the evaluation. The call p() is evaluated first. If it fails, the empty string is evaluated next, orempty() suspends with the empty string, and what happens next is pretty clear. Granted, it isn’t necessary for orempty() to suspend in this case, but it’s much easier (and less error-prone) just to plug the matching expression, as-is, in as the argument of suspend. If p() produces a result, orempty() suspends with that result — it does just what p() alone would have done. If orempty() is resumed, p() is resumed, since it suspended (being a matching expression). That’s all there is to it; everything else follows from what you already know.

Being able to pass matching procedures as arguments to other matching procedures opens up all kinds of possibilities. But if you look closely — or if you try some examples of your own, you’ll find one problem: Suppose a matching procedure you want to pass to another matching procedure has arguments of its own.

If you know there will be an argument, you can provide for it, as in:

```
procedure orempty(p,x)
  suspend p(x) | ""
end
```

But suppose the procedure has a lot of arguments. You could provide for as many as you’ll know you need. Perhaps it’s eight:

```
procedure orempty(p,x1,x2,x3,x4,x5,x6,x7,x8)
    suspend p(x1,x2,x3,x4,x5,x6,x7,x8) | ""
end
```

If p() doesn’t need eight arguments, the extra ones really don’t matter. But there’s always the possibility that later you may need more than eight arguments. More to the point, this kind of construction is just plain ugly.

You can use Icon’s variable-argument form of procedure declaration, coupled with list invocation, to get a general formulation:

```
procedure orempty(p,x[ ])
  suspend p!x | ""
end
```

The first argument to orempty() is the matching procedure. Any remaining arguments when orempty() is called are passed in a list, which is the value of x. The expression p!x calls p() with the arguments from the list x — just what’s needed.

We’ll end this article by expanding on an example from the second edition of *The Icon Programming Language* that shows the power of recursion.

Sometimes a particular construction in a string — a pattern — occurs several times in a row; perhaps zero or more. Suppose p() matches this pattern. What we want is a matching procedure arbno(p) that matches what p() matches, zero or more times in a row. Roughly speaking, what’s needed is

```
"" | p() | (p() || p()) | (p() || p() || p()) | …
```

Of course, there must be some context that causes the alterna-

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per copy for airmail postage to other countries.

tives to be evaluated.

Such an open-ended construction suggests a closed form, using recursion:

```
procedure arbno(p)
   suspend "" | (p() || arbno(p))
end
```

If you're accustomed to using recursion as a programming tool — and it's a powerful one indeed — you'll quickly see what's going on. Roughly stated, arbno(p) first matches the empty string (zero occurrences of p()). If it's resumed, it matches p() followed by arbno(p) — which matches the empty string first, so this amounts to one match of p(). And so on. If this informal argument is not clear, try writing out the result sequence for arbno(p). You might also experiment with an example and use procedure tracing to see what's actually happening.

We've again deliberately ignored the case where p() has arguments. We can use the same idea we used for orempty(), but the formulation is a little more difficult, since both p() and arbno(p) have to be called. The way to do it is to pass all the arguments to arbno() is a single list:

```
procedure arbno(x[])
   suspend "" | (x[1]!x[2:0] || arbno!x)
end
```

Consider an example:

```
arbno(move,1)
```

The argument x is the list [move,1]. Plugging this into the procedure body gives

```
suspend "" | (move![1] || arbno![move,1])
```

move![1] is the same as move(1) and arbno![move,1] is the same as arbno(move,1).

You may think this formulation is elegant or absurd, depending on your preference in programming style. We like it because it provides a general method of implementing a powerful programming facility.

More to Come

We've just scratched the surface of what can be done with string scanning when it is viewed in terms of pattern matching.

In an upcoming article, we'll explore how matching procedures can be used to model the powerful pattern-matching facilities of SNOBOL4. We'll also have something to say about relaxing the pattern-matching protocol so that expressions don't have to return the matched portion of the subject. This opens up the kinds of expressions that can be used and also can be used for string synthesis.

Gedanken Debugging

Gedanken experiments have been used very successfully to obtain insights into the workings of our physical universe. The word "gedanken" means "thought" in German. Albert Einstein's gedanken experiments are best known. As Roger Penrose aptly puts it [1], "In a thought experiment, one strives to discover general principles from the mere mental consideration of experiments that one might perform."

Gedanken debugging is based on a similar idea — trying to discover the cause and location of an error in a program by thinking about the program and what might occur if you tried various things, but without actually doing them.

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.

(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

Since there's no debugger for Icon (at least not yet), you aren't tempted as you might be in C to launch a debugger before even thinking about what's happened.

In Icon, you can turn on tracing or add expressions to produce diagnostic output. But you might think about the problem a little first — try some gedanken debugging. We contend that's a good idea even for programming languages with powerful debuggers.

Of course, gedanken debugging is a lot easier if you have considerable experience with programming Icon and tracking down bugs.

Here are a few suggestions that you may find useful in gedanken debugging:

• If the bug causes error termination, look carefully at the trace output. Look at all of it.

• Start by believing in the correctness of the diagnostic information that Icon provides. While Icon itself *may* have bugs, it's been used by many programmers for a long time and most known problems have been fixed. A very high percentage of program malfunctions are due to programming mistakes, not to problems with Icon itself.

• If the problem is at a known location, look at the code there carefully; don't just brush it off because it looks right at first glance. Give special consideration to possible syntactic pitfalls as listed in Issue 2 of the Analyst.

• If traceback information shows a null value where one should not be, look for a misspelled identifier or an uninitialized variable. And look for more than one null value while you're at it.

• Look at the nature of the symptoms and review programming pitfalls that may produce such problems.

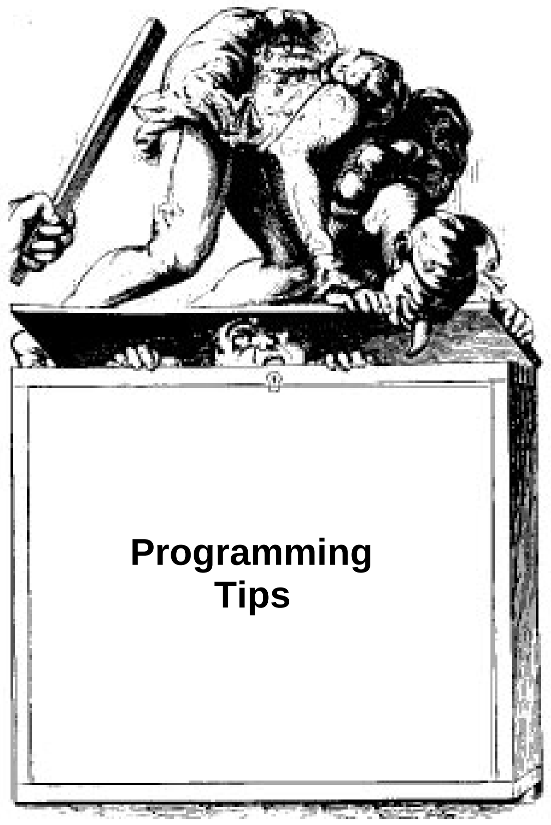
• If the bug is one that's just cropped up in a formerly working program, think carefully about changes you may have made recently. Don't dismiss a "trivial" change that "couldn't possibly cause a problem". Think about syntactic pitfalls in this regard. If that doesn't do it, retrace your steps (mentally). Maybe a change made a couple of days ago is just now causing problems.

• If the bug is particularly stubborn, give gedanken debugging some time, even overnight. Your subconscious may ferret out the problem if given the opportunity.

Of course, many problems require a combination of techniques, such as gedanken debugging in combination with tracing and selective diagnostic output. The point is, *think* about the problem; don't just rely on mechanical devices.

Reference

1. Roger Penrose, *The Emperor's New Mind*, Oxford University Press, 1989. p. 360.



Programming Tips

There may not be many occasions when you need to take advantage of the fact that Icon functions and procedures are "first-class" values that can be assigned to variables, passed to and from procedures, and so forth. It's actually more likely that you'll get into trouble because of this feature. For example,

```
tab := 8
```

wipes out the initial value of tab, which is (was) an important function.

Functions and procedures are no different in this regard, so we'll just use the term procedure for both here.

There are some neat, if somewhat obscure, things you can do with procedures as first-class values. For example,

```
if \output then Write := write else Write := 1
```

assigns the function for writing to Write, provided output is non-null, but assigns the value 1 to Write otherwise. (Write and write are different identifiers, of course — case is significant in Icon identifiers.)

If it's not clear what this might be good for, consider a program in which you want to be able to turn off all output selectively, perhaps under control of a command-line option. Here you can use Write for all output; if its value is the function for writing, output occurs as usual, but if its value is 1, the result is the first argument instead. Except in unusual circumstances, this is a "no-op".

If you're brave (or foolhardy), you could do this instead:

```
if /output then write := 1
```

That way, you don't have to use Write in your program and an existing program can be easily adapted. Of course, if you *do* need to write something, you've lost the function (unless you saved it somewhere first).

In fact, the programs in Icon's benchmark suite do something much like this (being careful to take care of writes also). When doing benchmarks, output is suppressed so that it doesn't skew the timings, but to be sure the program is working correctly, the output can be enabled without having to modify the programs.

Things like this work well for procedures, but what about operators like + ? Operators don't have names — that is, unlike functions, an operator is not the initial value of a global identifier. In fact, an operator is not a first-class value at all. Or is it? Is Icon hiding something?

Actually operators are values just like procedures, but without associated global identifiers. But is there a way to get at these values?

There's a modestly obscure function in Icon, proc(p), that can be used to determine if p is a procedure (which includes functions, of course). This function succeeds if p is a procedure but fails otherwise. If it succeeds, it returns the procedure. For example,

```
proc(system)
```

succeeds if system() is a supported function. The argument to proc() also can be the string name of a procedure, so you can do things like

```
if p := proc(read()) then p()
```

The interesting thing is that proc() works for operators too. Operators are given by their string names (like "+") and a second argument is used to indicate the number of arguments. (Some symbols, such as + , are used for both unary (prefix) and binary (infix) operators.) For example,

```
sum := proc("+",2)
```

assigns the infix addition operator to sum. Subsequently,

```
sum(i,j)
```

produces the sum of i and j.

So there really is a way to get at operators as values. It's closely associated with string invocation and there are a few limitations and restrictions (see the second edition of *The Icon Programming Language* for details).

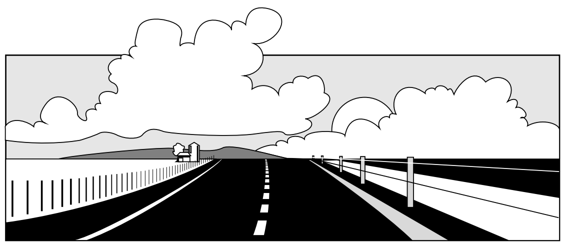
You still may not see any good reason for all this. The uses *are* esoteric. But keep the capability in mind. Maybe it will be just what you need someday.

Subscription Renewal

The next issue of the Analyst will complete the first year of publication.

Most present subscriptions are for one year. We'll include a renewal form in the next issue for those subscriptions that are expiring. You also can renew now to be sure of uninterrupted delivery.

See the box on page 10 of this issue for information.



What's Coming Up

Next time we'll approach string scanning from a different viewpoint — seeing how it really works by writing Icon procedures that model it.

We'll turn our attention to programming with structures with the first of a series of articles on how to program with structures. This first article will cover the role of pointers in Icon.

While everyone knows how to get an Icon program to stop running, there are some fine points that you may not have thought about. We'll have an article on program termination to address this issue.

And we'll have an article called "Evaluation Sandwiches". We'll leave the contents of that article to your imagination, but don't plan on a gastronomic delight.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

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