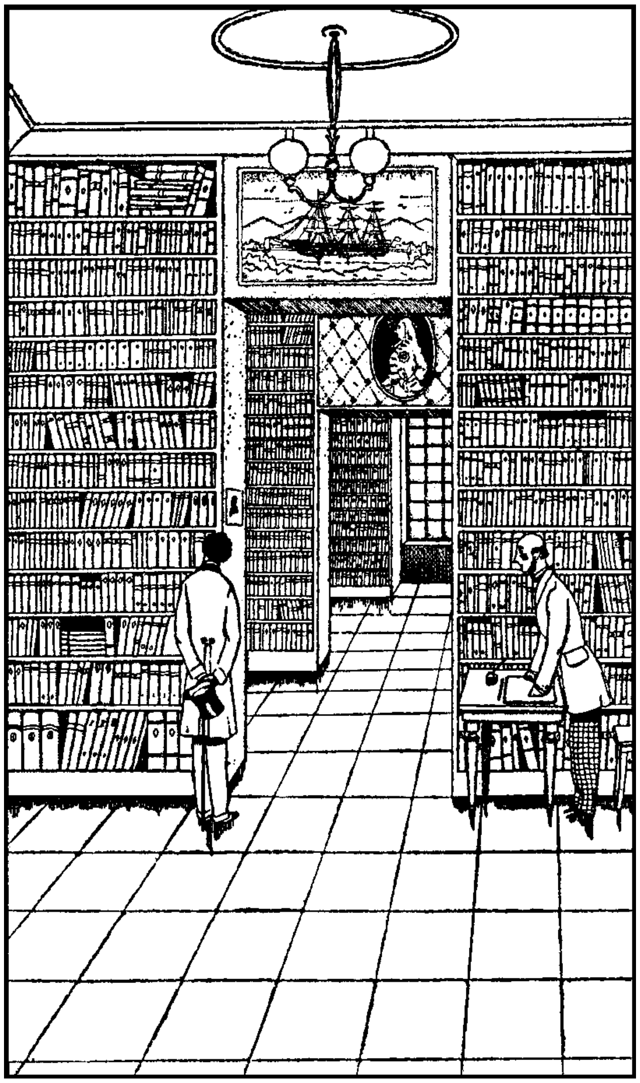
The Icon Analyst

In-Depth Coverage of the Icon Programming Language

August 1994
Number 25

In this issue ...



From the Library

Most programmers enjoy writing programs — at least some of the time. But not many programmers enjoy the inevitable debugging of a new effort, not to mention the feeling on discovery of a serious bug in a program written long ago.

Most programmers spend a lot of time debugging. How debugging is done and how hard it is depends on many things, not the least of which is the programming language used. Generally speaking, the higher the level of the language, the easier debugging is. But both the nature of bugs and the process of debugging are somewhat idiosyncratic to the language. Some language features may increase the likelihood of certain kinds of bugs. We've discussed such aspects of Icon in past articles in the Analyst.

Sometimes the most powerful and useful features of a language are the ones that cause the most trouble. Icon's sophisticated data structures are a case in point.

Data Structures

Icon's use of pointer semantics for structures makes it easy to do many things [1,2] and makes some kinds of operations efficient. A structure value, whether it's a record, list, set, or table, is just a pointer to (the memory address of) the block of elements that comprise the structure. Consequently, structure values are as compact as integers. The assignment of a structure value to a variable is as simple and as quick as the assignment of an integer. Icon's facilities for displaying structure values are limited, however. The function type(x) gives you the type of x. For example if x is a list, type(x) produces "list". That's handy and not something you can do in C, for example. You can get a little more information with image(x). For a list, it produces something like "list_10(3)". The 10 indicates that it's the tenth list created since the beginning of program execution. The 3 indicates the list has three elements, but it doesn't give a hint as to the nature of those elements, which might themselves be structures.

You can tell if two structures (pointers) are the same or not. For example, for structures x1 and x2

```
x1 === x2
```

compares pointers and succeeds if and only if x1 and x2 are *identical*. This operation is no help at all in determining if x1 and x2 are different structures but with the same values.

You also can't print a pointer, even though it's just a number. It's not that there's anything inherently difficult about printing the numerical value of a pointer. The problem is that garbage collection may move the elements to which a structure points, and hence change the pointer. If you were able to get the numerical value of a pointer, you would not be able to tell later on if it still pointed to the original elements for that structure.

There's also no direct way to show what the elements of a structure are; that's something you have to write yourself.

In cases where the elements of structures are pointers to other structures, debugging can be difficult.

Help from the Library

When Icon itself doesn't provide a facility you need, the first place to look for help is in the Icon program library. Several persons have contributed library procedures that show the details of structures. The best of these are in ximage.icn, written by Bob Alexander.

The procedure ximage(x) produces a string that describes x. If x is a structure, it shows the structure and its elements, and if an element is itself a structure, it shows that structure and so on. The result produced by ximage() resembles Icon code and hence is easy for Icon programmers to understand. Indentation and newlines are provided, so that if the result of ximage() is written, the output is nicely formatted.

It's easier to show what ximage() produces than it is to describe it. Suppose a program contains the following lines of code:

```
source := table()
basis := list(6, 0)
filter := list(10)
basis[1] := filter
basis[2] := basis
filter[3] := basis
source["basis"] := basis
source["filter"] := filter
```

Here's what write(ximage(source)) produces:

```
T1 := table(&null)
   T1["basis"] := L1 := list(6,0)
      L1[1] := L2 := list(10,&null)
         L2[3] := L1
      L1[2] := L1
   T1["filter"] := L2
```

Several things about this output are worth noting. One is that each structure is given a name (tag). The first letter of the tag indicates its type, with the number following producing a unique identification. The value of each structure is shown in the style of assignment as a strucure-creation function with its predominant element. (A table is shown with its default value). For example, most of the elements of basis (L1) are 0, while most of the elements of filter (L2) are null. Only the elements that are different from the predominant element are shown below the structure. The result is a compact but easily understood representation of structures.

Since every structure has a unique tag, pointer loops present no problem. For example,

```
node1 := []
node2 := []
put(node1, node2)
put(node2, node1)
put(node2, node2)

write(ximage(node1))
```

produces

```
L1 := list(1)
   L1[1] := L2 := list(2)
      L2[1] := L1
      L2[2] := L2
```

In addition to ximage(), there is a procedure xdump(x1, x2, …, xn) that applies ximage() to x1, x2, …, xn in succession and writes the results to standard error output. For example,

```
xdump("The basis:", basis)
```

writes

```
"The basis:"
L1 := list(6,0)
   L1[1] := L2 := list(10,&null)
      L2[3] := L1
```

```
L1[2] := L1
```

to standard error output.

We have come to adding

```
link ximage
```

to most of our programs as a matter of habit, using ximage() or xdump() when we get into trouble with structures, as we invariably do in complex programs.

Try it out; it can be a big help in understanding complex structures as well as in debugging them.

The Code

Generally speaking, we won't attempt to show code from the library; it often is too long to present in the context of the Analyst. The code for ximage() and xdump(), however, is relatively short and worth study.

Before going to the code, we'd like to point out that much of the value of ximage() is in its design. It produces an elegant result and unless you'd already seen it, you might not think of the approach that Bob used. At least we didn't; our version of a procedure to display structures is in the library. We left it there when Bob contributed his, in case someone was relying on it. But ours is not as nearly as good as Bob's.

Even given the specification for ximage(), writing such a procedure is a challenging task that is particularly difficult to do correctly. It includes not just producing error-free code, but also considering all the possibilities that need to be handled.

A listing of ximage() and xdump(), shorn of the library header and documentation, is shown opposite and on the next page. We've removed a few blank comment lines to get the whole listing on two pages.

We won't attempt to describe all the details; instead we suggest that you read though the code your-

```
procedure ximage(x,indent,done)
  local i,s,ss,state,t,xtag,tp,sn,sz
  static tr

  #  If this is the outer invocation, do some initialization.

  if /(state := done) then {
    tr := &trace ; &trace := 0    # postpone tracing while in here
    indent := ""
    done := table()
    }

  #  Determine the type and process accordingly.

  indent := (if indent == "" then "\n" else "") || indent || "    "
  ss := ""
  tp := type(x)
  s := if xtag := \done[x] then xtag else case tp of {

    #  Unstructured types just return their image().

    "null" | "string" | "integer" | "real" | "cset" | "co–expression" |
      "file" | "procedure" | "window" | "external": image(x)

    #  List.

    "list": {
          image(x) ? {
            tab(6)
            sn := tab(find("("))
            sz := tab(0)
            }
          done[x] := xtag := "L" || sn

          #  Figure out if there is a predominance of any object in the
          #  list.  If so, make it the default object.
          #
          t := table(0)
          every t[!x] +:= 1
          s := [,0]
          every t := !sort(t) do if s[2] < t[2] then s := t
          if s[2] > *x / 3 & s[2] > 2 then {
            s := s[1]
            t := ximage(s,indent || "    ",done)
            if t ? (not any('\'"') & ss := tab(find(" :="))) then
                    t := "{" || t || indent || "    " || ss || "}"
            }
          else s := t := &null

          #  Output the non–defaulted elements of the list.

          ss := ""
          every i := 1 to *x do if x[i] ~=== s then {
            ss ||:= indent || xtag || "[" || i || "] := " ||
                    ximage(x[i],indent,done)
            }
          s := tp || sz
          s[–1:–1] := "," || \t
          xtag || " := " || s || ss
          }

    #  Set.

    "set": {
          image(x) ? {
            tab(5)
            sn := tab(find("("))
            }
          done[x] := xtag := "S" || sn
          every i := !sort(x) do {
            t := ximage(i,indent || "    ",done)
            if t ? (not any('\'"') & s := tab(find(" :="))) then
                    t := "{" || t || indent || "    " || s || "}"
```

self to understand Bob’s approach and coding techniques. A few points deserve note, however.

One is the use of two extra arguments in ximage() in addition to the one the user provides: indent and done. As you’d expect, ximage() calls itself recursively. The argument indent, which is null initially because it is omitted in the top-level call, keeps track of the indentation, which depends on the level of recursion. The argument done, also null initially, is a table that contains the values and their names (tags). The use of additional arguments to pass on values and make them accessible to all levels of recursive calls is described in more detail in Reference 3.

The main part of ximage() is a case expression that handles different types. The first case clause handles unstructured types, for which Icon’s built-in function image() is used. The next case clauses handle lists, sets, and tables. Records present a special problem, since each record declaration produces a separate type. Records can be handed in the default clause, since all other types are handled in previous case clauses. (That’s why unstructured types are handled in a previous clause, rather than as an afterthought in the default clause.)

The procedure xdump() takes variable number of arguments and applies ximage() to each one.

One thing about these procedures that you should note is the care that has been taken with fine points. For example, ximage() turns tracing off during its invocation and restores it when it is done. The procedure xdump() goes to extra care to return its last argument, following the design of the functions write() and writes(). xdump() even is careful to return and not fail if it’s called with no arguments.

```
               ss ||:= indent || "insert(" || xtag || "," || t || ")"
               }
            xtag || " := " || "set()" || ss
            }

      #  Table.

      "table": {
            image(x) ? {
               tab(7)
               sn := tab(find("("))
               }
            done[x] := xtag := "T" || sn

            #  Output the table elements.  This is a bit tricky, since
            #  the subscripts might be structured, too.

            every i := !sort(x) do {
               t := ximage(i[1],indent || "   ",done)
               if t ? (not any('\'"') & s := tab(find(" :="))) then
                        t := "{" || t || indent || "   " || s || "}"
               ss ||:= indent || xtag || "[" ||
                        t || "] := " ||
                        ximage(i[2],indent,done)
              }

            #  Output the table, including its default value (which might
            #  also be structured.

            t := ximage(x[[]],indent || "   ",done)
            if t ? (not any('\'"') & s := tab(find(" :="))) then
                 t := "{" || t || indent || "   " || s || "}"
            xtag || " := " || "table(" || t || ")" || ss
            }

      #  Record.

      default: {
            image(x) ? {
               move(7)
               t := ""
               while t ||:= tab(find("_")) || move(1)
               t[-1] := ""
               sn := tab(find("("))
               }
            done[x] := xtag := "R_" || t ||  "_" || sn
            every i := 1 to *x do {
               name(x[i]) ? (tab(find(".")),sn := tab(0))
               ss ||:= indent || xtag || sn || " := " ||
                        ximage(\x[i],indent,done)
               }
            xtag || " := " || t || "()" || ss
            }
      }

   #  If this is the outer invocation, clean up before returning.

   if /state then {
      &trace := tr                    # restore &trace
      }

   #  Return the result.

   return s
end

#  Write ximages of x1,x1,...,xn.

procedure xdump(x[])
   every write(&errout,ximage(!x))
   return x[-1] | &null
end
```

Conclusions

We keep promoting the Icon program library because we know it's an excellent resource for Icon programmers. The library contains a few gems like ximage as well as some programs and procedures that, well, are not so good. We'll continue to explore the library in future issues of the Analyst, presenting some of the most useful items.

For all the elegance of ximage(), you probably would prefer a procedure that draws connecting lines between structures, especially for platforms that support graphics.

Some time ago, before Icon had graphics facilities, Roger Hayes, a student in one of our classes on string and list processing, produced a package for diagramming structures with connecting lines represented by dashes and vertical bars. It was cleverly done, but hopelessly difficult to understand except for the simplest cases. More recently, Song Liang wrote a structure visualization package as a class project [4]. It provides a window for each structure, and allows scrolling for structures with many elements. It does not, however, show any connections between structures.

There are serious problems with diagramming structures with connecting lines. Laying out an arbitrary graph in an understandable way is an essentially intractable problem. If there are hundreds or thousands of structures, or if a structure has thousands of elements that point to other structures, there's no way to fit everything on the screen at one time. And even if that were possible, the display probably wouldn't be understandable.

There are various approaches to working around these problems, such as miniaturization, facilities for navigating through "structure space", "fish-eye views" that enlarge areas of interest, and so on. But these are research problems that have been and will continue to be topics of research and many doctoral dissertations.

Still, it would be nice to have something that is useful for simple cases. Any takers? Bob?

References

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 189-199.

2. "Pointer Semantics", The Icon Analyst 6, pp. 2-8.

3. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 194-195.

4. "Icon Class Projects", *Icon Newsletter* 40, pp. 5-6.

Anatomy of a Program — Lindenmayer Systems

One of the advantages of using Icon is that many programming tasks are easy — easy enough to warrant undertaking projects you might not want to try in a lower-level language, such as C. This is particularly true of problems related to formal languages, which often depend heavily on string manipulation.

In an earlier Analyst, we described how to produce recognizers for context-free languages [1]. Here we'll look at a different kind of formal grammar that was developed by Aristid Lindenmayer, a Hungarian botanist.

A Formalism for Plant Development

Lindenmayer's interest was in formal models of plant development. In his system, each plant cell is represented by a symbol (character), and the development of a plant goes through a series of "generations" in which each symbol is replaced by other symbols according to precisely defined rules.

A model of a primitive alga has two kinds of cells, which can be represented by C and D (the actual characters used are irrelevant). The rules for replacement are:

1. Every C is replaced by DC.
2. Every D is replaced by C.

If we start with CD (the "axiom"), we get CDC, the original C being replaced by CD and the original D being replaced by C. In subsequent generations, CDC is "rewritten" as CDCCD, then CDCCDCDC, and so on.

Formal grammars like this are called Lindenmayer systems, or L-systems for short. For simple plants, at least, L-systems can be quite successful in characterizing development, and much work has been done on the subject [2-4]. Not surprisingly, such a simple view of plant development has its limitations. Nonetheless, it's sufficiently interesting to see what can be done with a little programming.

It's worth pointing out before going on that there is a fundamental difference between L-systems and context-free grammars, although the two look similar. In a context-free grammar there may be several possible replacements for a nonterminal symbol, at each rewriting only one nonterminal symbol is replaced, and replacing every possible nonterminal symbol in all possible ways leads to many strings. In an L-system, there is no distinction between terminal and nonterminal symbols (or every symbol can be thought of as a nonterminal symbol), there is only one possible replacement for a symbol, and *every* symbol is replaced at each rewriting (generation). These different rules are what give L-systems their descriptive power. This may seem contradictory, since there are so many more possible strings in context-free grammars. But all the possibilities in a context-free grammar make it impossible to exclude "unwanted" strings [5]. For example, if the rules given earlier are applied as context-free replacements, CD leads to two new strings, CCD and CC. At the next step there are five strings, CDCD, CCDD, CCC, CCD, and CDC, and so on. It's possible to construct a context-free grammar that includes the same strings that an L-system produces, but such a context-free grammar inevitably includes many more strings than the L-system does.

A Program for L-Systems

It's easy to write an Icon program to produce the result after a given number of generations for a specific L-system. (It's just as easy to produce the results of each generation, but the result at a specific generation usually is what's of interest.)

A table is the obvious way to represent the rules, as in

```
rule := table()

rule["C"] := "CD"
rule["D"] := "C"
```

Code to produce gener generations is simple:

```
current := "CD"          # axiom

every 1 to gener do {
  new := ""
  every new ||:= rule[!current]
  current := new
  }

write(current)
```

Of course, what we really want is a program to read in an L-system and produce the result of a specified number of generations. To do this, we need a syntax for describing L-systems. We'll use one in which the axiom and the number of generations are given by a name followed by a colon and the value, while a replacement rule is given with "–>" separating the symbol to be replaced from its replacement. The L-system given above for five generations looks like this:

```
axiom:CD
gener:5
C–>CD
D–>C
```

Code to read in such an L-system is relatively simple, Here's one version:

```
while line := read() do
  line ? {
    if sym := tab(find("–>")) then {
      move(2)
      rule[sym] := tab(0)
      }
    else if keyword := tab(find(":")) then {
      move(1)
      value := tab(0)
      case keyword of {
        "axiom":  axiom := value
        "gener":  gener := value
        default:
          stop("*** invalid line: ", line)
        }
      }
    else stop("*** invalid line: ", line)
    }
```

Putting this together with the rewriting loop given earlier is all that's needed for a complete program.

There's another aspect of L-systems that we haven't mentioned: There may be symbols for which there are no replacement rules. Such symbols don't represent cells, but they are needed to model some kinds of plant development. A symbol for which there is no replacement rule is left unchanged during replacement (or replaced by itself, if you prefer). For example, if the axiom is CXD, the replacement rules given earlier produce CDXC for the first generation.

Symbols for which there are no replacement rules complicate the program a bit. Two possibilities are (1) recognize a symbol that is not in the table of replacements in the rewriting loop and replace it by itself, or (2) put each such symbol in the table of replacements with itself as its value.

For the first alternative, it's easy enough to recognize a symbol for which there is no replacement, since subscripting the table with it produces the default null value of the table. The rewriting loop can be done as follows:

```
every 1 to gener do {
  new := ""
  every sym := !current do
    new ||:= (\rule[sym] | sym)
  current := new
  }
```

The alternative approach of putting all symbols in the table before rewriting is more attractive, at least from the point of view of efficiency, but it's also more complicated to program. As it turns out, the little bit of extra work in the rewriting loop doesn't matter much, but we'll show how to get replacements of all symbols in the table anyway. The idea is simply to keep track of all symbols in the axiom and replacement rules while the L-system is being read, and add "identity" replacements for which there are no replacement rules before starting the rewriting:

```
rule := table()

allsyms := ''          # initially empty cset

#  Read L-system and get all symbols.

while line := read() do
  line ? {
    if sym := tab(find("->")) then {
      move(2)
      replace := tab(0)
      rule[sym] := replace
      allsyms ++:= replace
      }
    else if keyword := tab(find(":")) then {
      move(1)
      value := tab(0)
      case keyword of {
        "axiom":  {
          allsyms ++:= value
          axiom := value
          }
        "gener":   gener := value
        default:
          stop("*** invalid line: ", line)
        }
      }
    else stop("*** invalid line: ", line)
    }

# Now add identity replacements for all
#  symbols not in the table.

every sym := !allsyms do
  /rule[sym] := sym
```

Since the operator that tests for a null value returns a variable if its argument is a variable, an assignment can be made in the same expression as the test.

Now the original rewriting loop can be used without testing every symbol to see if there's a replacement for it.

There's a middle ground between the two possibilities discussed above: Don't fill in the table after reading the L-system, but add a symbol for which there is no replacement to the table when that symbol is first encountered during rewriting:

```
every 1 to gener do {
  new := ""
  every sym := !current do
    new ||:= (\rule[sym] | (rule[sym] := sym))
  current := new
  }
```

A Problem

So far, so good; any of the three approaches to handling symbols for which there are no replacement rules will work, and the program can read in and generate the result for any well-formed L-system.

There's a lurking problem, however. For L-systems of interest, the strings for successive generations get longer and longer. For the simple L-system that we've been using as an example, the strings do not grow in length very rapidly. But for most L-systems, they do. Consider this one:

```
gener:2
axiom:A
A->BCDDAEFAEFBDFBAECA
B->BB
```

The second-generation for this string is:

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

cs.arizona.edu (cd /icon)

```
BBCDDBCDDAEFAEFBDFBAECAEFBCD
DAEFAEFBDFBAECAEFBBDFBBBCDDA
EFAEFBDFBAECAECBCDDAEFAEFBDFB
AECA
```

and the 10th-generation string has over six million characters. That would exceed the capacity of many computers. This problem, in fact, limited early work on L-systems.

An Alternative Approach

There is a solution to this problem, and an elegant one. It requires a bit of insight and leads to an interesting coding technique.

The insight is that it's possible to go through all the generations for the first symbol of the axiom before going on to the second and subsequent symbols. Then each symbol in the last generation can be written before going on to the next. Of course, replacing any symbol may produce several symbols, but only the first of these is carried to the last generation, while the other symbols are (conceptually) prepended to part of the string that has not yet been processed.

Using this approach, there is no need for concatenation and no strings build up in memory.

Most of the time when we come across "an interesting coding technique" in Icon, it involves generators or recursion, or both [6]. It's fairly easy to imagine generating the symbols; how to generate them in the right order is less obvious. Rewriting for a specified number of generations is done iteratively in the examples presented earlier. With a more complex requirement, it is easier to use recursion. Here's a recursive procedure for generating the symbols in the order needed:

```
procedure lgen(sym, rule, gener)

  if gener = 0 then return sym
  suspend lgen(!rule[sym], rule, gener – 1)

end
```

The current symbol is sym, rule is the table, and gener is the number of generations *remaining* to be done.

When gener reaches 0, the current symbol is returned, ending the recursion. Otherwise, lgen() is called recursively for each symbol in the replacement for the current one, but with one less generation to go. (This formulation assumes there are replacements in the table for all symbols, but it's easy enough to recast it for the case where that's not true.)

That's all there is to it, except to use writes() to write the symbols on one line as they are generated:

```
every writes(lgen(!axiom, rule, gener))
write()                      # terminate line
```

We're rather proud of the recursive generator shown above. We developed it while improving a rather ancient L-system program in the Icon program library. It's worth noting, however, that there's nothing magical going on. After developing our procedure, we discovered that Steve Wampler had used a very similar technique in his version of an L-system program. You'll also find similar recursive generators in other Icon programs.

We wish we had a deep understanding that would enable us to describe how to arrive at such a procedure in a methodical and mechanical way. We don't, but we're working on it. Our advice in the meantime is to "think generators" and "think recursion". Better yet, think "recursive generators".

The Complete Program

So far, we've shown bits and pieces of programs. So that you can see a complete program, here's the one that generates symbols one at a time:

```
procedure main()
  local rule, line, sym, new, axiom, gener
  local allsyms, replace, keyword, value

  rule := table()

  allsyms := ''       # initially empty cset

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

      if /axiom then stop("*** no axiom")
      /gener := 5         # default

      every sym := !allsyms do
        /rule[sym] := sym

      every writes(lgen(!axiom, rule, gener))
      write()             # terminate line

    end

    procedure lgen(sym, rule, gener)

      if gener = 0 then return sym
      suspend lgen(!rule[sym], rule, gener – 1)

    end
```

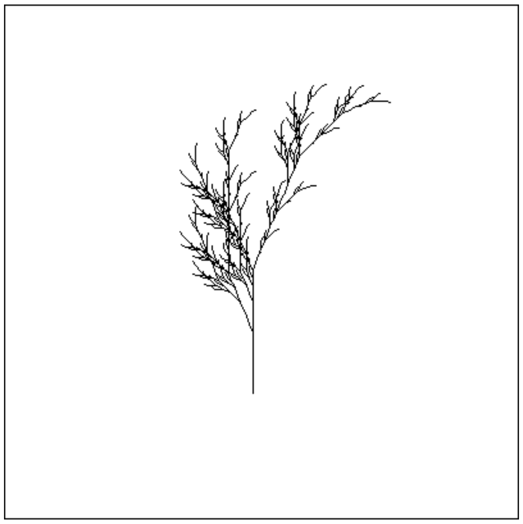
Conclusion

Aside from the claim that L-systems can be used to model the development of simple plants, the results produced by our program are just strings of essentially meaningless symbols — the result of applying replacement rules to the axiom in a fixed way. In fact, many L-systems have nothing to do with plants and can be entirely arbitrary.

Lindenmayer himself originally consider L-systems only as a formal approach to plant development. Two of his graduate students [7] had an idea that turned out to have startling ramifications: using the strings produced by L-systems to actually draw the plants being modeled.

Their method involved interpreting specific symbols as having graphical meaning that could be expressed by drawing operations — drawing operations for which turtle graphics [8] are ideally suited.

We'll explore this subject in the next issue of the Analyst. For now, we'll leave you with the image below.



References

1. "The Anatomy of a Program — A Recognizer Generator", The Icon Analyst 10, pp. 4-9.

2. *Formal Languages*, Arto Salomaa, Academic Press, New York, 1973, pp. 234-252.

3. *Developmental Systems and Languages*, Gabor T. Herman and Grzegorz Rozenberg, North-Holland Publishing Company, New York, 1975.

4. *The Book of L*, Grzegorz Rozenberg and Arto Salomaa, Springer-Verlag, Berlin. 1986.

5. *An Introduction to Automata Theory, Languages, and Computation*, John E. Hopcroft and Jeffery D. Ullman, Addison-Wesley Publishing Company, Reading, Massachusetts, 1979, pp. 55-58, 125-128.

6. "Programming Tips — Recursive Generators", The Icon Analyst 13, pp. 10-12.

7. *Artificial Life: The Quest for a New Creation*, Steven Levy, Pantheon Books, New York, 1992, pp. 234-237.

8. "Turtle Graphics", The Icon Analyst 24, pp. 6-10.

From the Wizards

In an earlier article in this feature [1], we showed how a case expression could be used to select an expression to evaluate depending on whether a number was less than, equal to, or greater than another. That is, of course, just a special case of using comparisons in case expressions. Here we'll show how you can select an expression depending on the range into which it falls.

Suppose, for example, that you want to assign letter grades based on test scores. The conventional approach looks like this:

```
if score > 92 then grade := "A"
else if score > 80 then grade := "B"
else if score > 62 then grade := "C"
else if score > 50 then grade := "D"
else grade := "F"
```

Since if-then-else returns the selected expression, this can be written more compactly as

```
grade :=
  if score > 92 then "A"
  else if score > 80 then "B"
  else if score > 62 then "C"
  else if score > 50 then "D"
  else "F"
```

There is an alternative formulation takes advantage of the fact that a selector in a case expression can succeed or fail:

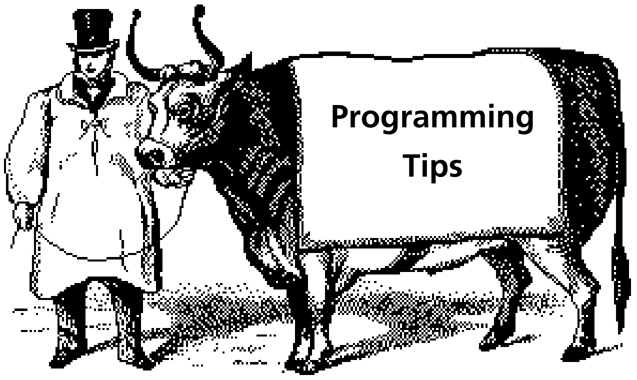
```
grade := case score of {
  (score > 92) & score:    "A"
  (score > 80) & score:    "B"
  (score > 62) & score:    "C"
  (score > 50) & score:    "D"
  default:                 "F"
  }
```

Case selectors are evaluated in order. If score is greater than 92, the first selector succeeds, the conjunction produces score, which matches the case value, and the result of the case expression is "A". If score is not greater than 92, the case selector fails, the next case selector is evaluated, and so on.

We grant that this kind of construction is a bit contrived. But it suggests some of the things that can be done by exploiting the evaluation of expressions in case selectors.

Reference

1. "From the Wizards", The Icon Analyst 18, p. 4-12.



Local Identifiers

A lot of attention is paid to debugging; perhaps not enough is paid to avoiding problems in the first place. This is the first of three programming tips that can help you prevent bugs in your Icon programs.

Icon takes a permissive attitude toward scope declarations. If you don't declare an identifier that you use in a procedure, its scope defaults to local, provided that there is no global declaration for it.

This saves keyboarding and works well most of the time. When it doesn't work, the kinds of bugs that result can be serious and sometimes difficult to track down.

Consider this simple program:

```
procedure main()
  every write(words())
end
procedure words()
  while line := read() do {
    line ? {
      while tab(upto(&letters)) do {
        word := tab(many(&letters))
```

```
            suspend word
            }
         }
      }
   end
```

The identifiers line and word in words() are undeclared locals. No problem. But suppose that you transpose two letters when keyboarding this program and enter

```
suspend wrod
```

instead of

```
suspend word
```

If you don't notice your mistake, words() will generate a sequence of null values and the program output will consist of blank lines.

You'll probably figure out the cause of this particular bug easily enough, but perhaps not until you've translated and run the program and examined the output.

This kind of problem is easily caught by using the –u option for the Icon translator. For example, if the program file is named wordlist.icn, the following will do:

```
icont –u wordlist
```

If you make the typing error mentioned above, the linker will produce the following output:

```
Translating:
wordlist.icn:
  main
  words
No errors
Linking:
wordlist.icn: "line":   undeclared identifier, procedure words
wordlist.icn: "word": undeclared identifier, procedure words
wordlist.icn: "wrod": undeclared identifier, procedure words
```

Your attention probably will be attracted to the appearance of both word and wrod in the messages, and you'll be able to correct your program before running it.

We recommend that all local identifiers be declared, even though Icon doesn't require it. This suppresses linker warning messages so any warning message means you've either forgotten to declare a local identifier or you have a local identifier you didn't expect. Sure, it takes a bit of extra work. In the case above, a line of the form

```
local line, word
```

is needed at the beginning of the procedure words(). But it's not that much work, and making it a habit can save you a lot of time in the long run. (We personally think the design of Icon is in error on this point and that, at least, the –u option to icont should have been the default. You can arrange that yourself with a simple script.)

There are subtler points about local identifiers. In the procedure words() given above, read, tab, upto, and many also are undeclared identifiers. Since they are the names of functions, global declarations for them are implicit and their interpretation within the procedure is what you expect.

The other side of the coin is that you can accidentally use the name of a function as an iden-

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

The Bright Forest Company
Tucson Arizona

tifier that's intended to be local. This can get you in big trouble. Consider

```
procedure count()
   tab := 0
   every words() do          # tabulate words
      tab +:= 1
   return tab
end
```

Here tab is used to keep a count of the words. Since tab is not declared to be local and tab is a function, the use of tab in this procedure refers to the global identifier. Assigning 0 to it wipes out the initial function value it had. When words() is called, the use of tab there refers to the global variable, but it's value is zero, not the expected function. All hell breaks loose, since

```
tab(upto(&letters))
```

is equivalent to

```
0(upto(&letters))
```

This expression is perfectly legal, if somewhat meaningless. If an integer is applied to an argument list, it selects the argument following that position. As is usual in Icon, a nonpositive specification is taken relative to the end of the list. Here, 0 means the argument after the last one, and the expression simply fails.

Consider the consequences in words(). The while loop in string scanning immediately fails, no words are generated, and count() always returns 0. This may be baffling, indeed.

The problem here would have been avoided if tab had been declared local in count(). Then it could have been used in count() without affecting the value of tab in words().

You might argue that using tab as a local identifier is bad practice. Indeed it is. But it's an easy mistake to make. We've even seen this in a program:

```
table := table()
```

That doesn't even cause a problem if table() is not called again.

When you write Icon programs, you learn to avoid local identifiers that are the names of functions. But there are some functions that are rarely used and that make inviting names for local identifiers. You may have seen this in some of our programs:

```
procedure main(args)
```

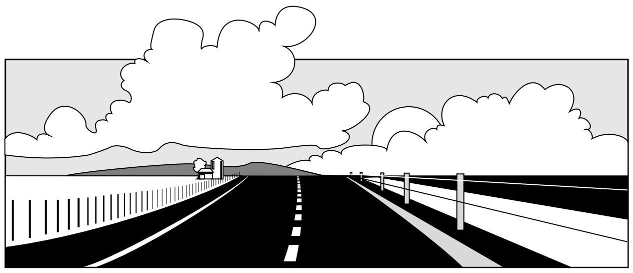
But there is a function args(). Fortunately, it's not used often and procedure parameters are automatically declared to be local. Nonetheless, we've tried to use args as a parameter name and as a function in the same procedure. You can imagine what happened.

We should mention one subtle point before ending this tip. If you want undeclared local identifiers in library procedures to get warning messages during linking, you need to use –u when you translate the procedures to get ucode, as in

```
icont –u –c wordlib
```

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 order for airmail postage to other countries.



What's Coming Up

In the next issue of the Analyst, we'll have another article on Lindenmayer systems, this time showing how to use turtle graphics to produce drawings from L-systems.

We have articles in the works on random numbers, symbolic mathematics, and string invocation. One of these probably will appear in the next Analyst.

We'll also have another tip on how to avoid bugs in Icon programs.