The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1994
Number 27

In this issue …

Anatomy of a Program — Lindenmayer Systems (continued)

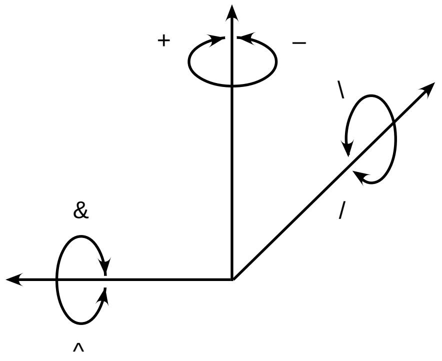
As we mentioned in the last issue of the *Analyst*, various kinds of L-systems have been developed to describe plants and plant development. Most of these L-systems are considerably more difficult to implement than 0L-systems, and we won't attempt them here. There are, however, generalizations that can be made to 0L-systems.

Generalizations to 0L-systems

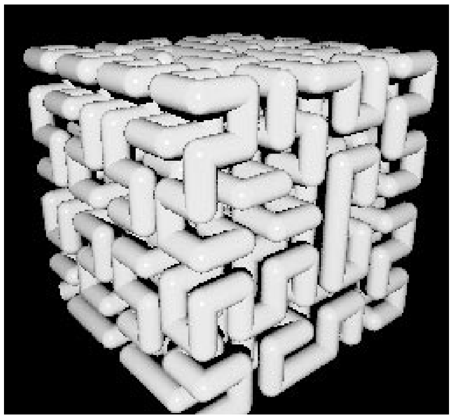
Most real plants develop in three dimensions, not just two. To handle this, roll and pitch can be added to the directional commands, so that the full set becomes:

+	turn right by δ
–	turn left by δ
&	pitch down by δ
^	pitch up by δ
\	roll left by δ
/	roll right by δ
|	turn around

where δ is the specified angle for turns:



As illustrated in the last article, L-systems can be used to characterize fractals as well as plants. Here's an example of a three-dimensional fractal that can be described by a 0L-system:



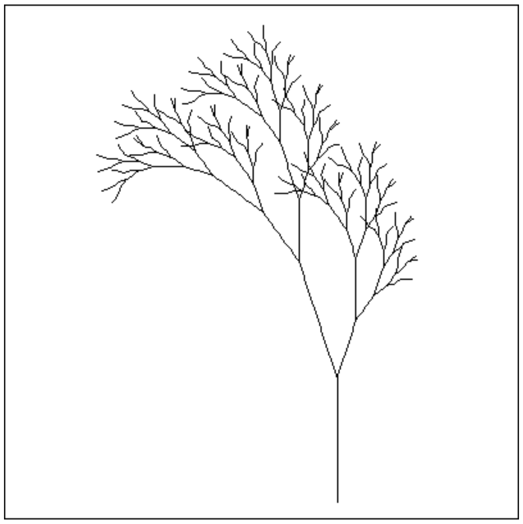
Our program can't produce such sophisticated renderings even in two dimensions, but it's interesting to have an idea of the capability of 0L-systems.

Two examples of three-dimensional plant 0L-systems are:

```
axiom:A
angle:30
A–>[B/////B///////B]
B–>[&ILA]
I–>FL
F–>F/////I
L–>[^^–F+F+F–|–F+F+F]

axiom:P
angle:18
P–>I+[P+O]—//[—L]I[++L]–[PO]++PO
I–>FS[//&&L][//^^L]FS
S–>SFS
L–>[+f–ff–f+|+f–ff–f]
O–>[&&&D/W////W////W////W////W]
D–>FF
W–>[^F][&&&&–f+f|–f+f]
```

Since the implementation of 0L-systems in the last article ignores symbols for which there are no replacements, we can use it as-is to draw a projection" of three-dimensional 0L-systems onto two dimensions, just ignoring turns in the third dimension. The second 0L-system above produces the following drawing when done this way:



We can't do much about three-dimensional rendering, since the Icon turtle graphics package is limited to drawing in two dimensions. If it worked in three dimensions, our program could provide three-dimensional drawings just by calling appropriate turtle graphics procedures. In any event, this capability lies outside our program, and until a three-dimensional turtle graphics package comes along, there's not much we can do about it. (Any offers?)

Before going on to generalizations that we can handle, here's the program from the last article for reference:

```
link turtle                  # turtle graphics package

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

  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
          "angle":   angle := real(value) |
            stop("*** invalid angle: ", line)
          "length":  length := integer(value) |
            stop("*** invalid length: ", line)
          default:
            stop("*** invalid keyword: ", line)
          }
        }
      else stop("*** invalid specification: ", line)
      }

  if /axiom then stop("*** no axiom")
  /length := 5               # defaults
  /gener := 4
  /angle := 90.0

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

  every sym := lgen(!axiom, rule, gener) do
    case sym of {
      "F":   TDraw(length)
      "f":   TSkip(length)
      "+":   TRight(angle)
      "–":   TLeft(angle)
      "[":   TSave()
      "]":   TRestore()
      }

  Event()                    # wait to dismiss window

end

procedure lgen(sym, rule, gener)

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

end
```

There are various extensions to 0L-systems that can be done within the current framework. An example is incrementally diminishing the thickness of "branches" as drawing progresses toward the extremities. All that's needed is a symbol to represent decrementing the thickness and adding the necessary initialization and drawing code.

But some kinds of extensions can't be made without a significant change to the notational system we've been using. For example, if you want to specify an arbitrary thickness instead of one obtained by decrementing the current thickness, there's no way to do it with the single-character symbols we've been using; our syntax is too impoverished. There are other problems with using only single-character symbols. Although we're not likely to run out of characters, that's at least a possibility. Furthermore, the use of single-character symbols makes L-systems difficult to understand and to keyboard correctly. Just being able to use multi-character symbols for our "place markers" would be a big help. For example, the first 0L-system on page 2 is much easier to understand when written this way:

```
axiom:apex
angle:30
apex –> [ branch / / / / / branch / / / / / / / branch ]
branch –> [ & internode leaf apex ]
internode –> F leaf
F –> F / / / / / internode
leaf –> [ ^ ^ – F + F + F – I – F + F + F ]
```

We've added blanks to separate the symbols now that they are not all single characters.

Usually when you buy into a notational system for its structural simplicity and ease of processing, it requires major changes to a program to go to a more general syntax. It's actually fairly easy to go from single-character symbols to multi-character ones in the programs we've given.

There is a simple but important idea in the conversion. Replacement rules in a 0L-system are sequences of *symbols*. So far, we've used single-character symbols in which a replacement is represented by a string, which is just a sequence of characters. Suppose now that we generalize the syntax of symbols to allow more than one character. The symbols then become strings. If we represent replacements by strings, as we will need to do for writing them down, separators are needed as shown in the previous example.

We could do this inside the program and parse the replacements during drawing. But there's a better way.

As noted above, a string is a sequence of characters and hence provides a natural and easily processed representation for single-character symbols. For multi-character symbols, there also is a natural representation: a *list* of *strings* (symbols).

With this idea in mind, we can look at our program to see what needs changing.

A bit of insight helps at this stage. Before getting involved with multi-character symbols, we can try the new data representation — lists instead of strings — with our present program. After all, a single-character symbol is just a special case of a multi-character symbol.

Consider the simple 0L-system from the last article:

```
axiom:X
X–>F–[[X]+X]+F[+FX]–X
F–>FF
```

Represented internally as a list of strings, the first replacement looks like this:

```
["F", "–", "[", "[", "X", "]", "+", "X", "]",
 "+", "F", "[", "+", "F", "X", "]", "–", "X"]
```

That would be a pain to keyboard, but fortunately, we don't have to; the program will read in strings as before and produce lists instead of strings for replacements.

It's easy enough to convert a string into a list of one-character strings:

```
rep := []
every put(rep, !s)
```

We can start by making this change to our earlier program. Instead of putting the code above in-line, we'll encapsulate it in a procedure instead, anticipating that we'll need something a bit more complicated when we convert to multi-character symbols:

```
procedure parse(s)
  local rep

  rep := []
  every put(rep, !s)

  return rep

end
```

With this, the line

```
rule[sym] := replace
```

is replaced by

```
rule[sym] := parse(replace)
```

so that the value of rule[sym] is now a list instead of a string. Similarly, the axiom needs to be put in a list:

```
axiom := [rule]
```

What else needs to be changed? Not much. The places in the program that process replacements only expect a data structure that is a sequence of *symbols*. Icon's element-generation operator, !x, applies to lists as well as to strings. For strings, it produces one-character substrings. For lists, it produces the elements (in this case, also one-character strings).

This is an excellent example of the value of polymorphism — allowing operations to work in similar manners on different types of data.

The one-line change above, along with the simple procedure parse(), is all that's needed to test the list representation for replacements. Only a little more is needed to go to multi-character symbols.

The procedure parse() needs to be changed to handle multi-character symbols separated by blanks:

```
procedure parse(s)
  local rep

  rep := []

  s ? {
    tab(many(' '))           # leading blanks?
    while put(rep, tab(upto(' ') | 0)) do
    tab(many(' ') | break
    }

  return rep

end
```

Possible blanks after "–>" also need to be removed from sym:

```
if sym := trim(tab(find("–>"))) then {
   ...
```

It's also necessary to change from csets to sets for keeping track of symbols:

```
allsyms := set()     # initially empty set
```

and

```
insert(allsyms, value)
```

instead of

```
allsyms ++:= value
```

That's it: minor changes to a few lines (some only to improve the code) and the addition of a simple procedure for parsing replacements. In summary, here is the new program. The changed lines are marked at the left.

```
    link turtle

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

      rule := table()
➻     allsyms := set()            # initially empty set

      while line := read() do
        line ? {
➻         if sym := trim(tab(find("–>"))) then {
            move(2)
➻           value := parse(tab(0))
➻           rule[sym] := value
➻           every insert(allsyms, !value)
            }
          else if keyword := tab(find(":")) then {
            move(1)
            value := tab(0)
            case keyword of {
              "axiom":  {
➻               axiom := parse(value)
➻               every insert(allsyms, !axiom)
                }
              "gener":  gener := value
              "angle":  angle := real(value) |
                stop("*** invalid angle: ", line)
              "length": length := integer(value) |
                stop("*** invalid length: ", line)
              default:
                stop("*** invalid keyword: ", line)
              }
            }
          else stop("*** invalid specification: ", line)
          }
      if /axiom then stop("*** no axiom")

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

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

      every sym := lgen(!axiom, rule, gener) do
        case sym of {
          "F":  TDraw(length)
          "f":  TSkip(length)
          "+":  TRight(angle)
          "–":  TLeft(angle)
          "[":  TSave()
          "]":  TRestore()
          }

      Event()              # wait to dismiss window

    end
```

```
procedure lgen(sym, rule, gener)
  if gener = 0 then return sym
  suspend lgen(!rule[sym], rule, gener – 1)
end
procedure parse(s)
  local rep
  rep := []
  s ? {
    tab(many(' '))
    while put(rep, tab(upto(' ') | 0)) do
    tab(many(' ')) | break
    }
  return rep
end
```

Yet Another Article

We have one more article on L-systems to finish this series. The final article will show an entirely different way of implementing 0L-systems — compiling the specification of a 0L-system into an Icon program instead of interpreting the 0L-system, as we have done so far.

Static Analysis of Icon Programs

Introduction

Icon is a large programming language as measured in terms of its computational repertoire. Standard Version of 9 Icon has 85 operators, 139 functions, 64 keywords, 17 control structures, and several other operations. Some of these are essential to programming and are found in some form in almost all programming languages. Examples are arithmetic and simple input and output. Other features are special to Icon and distinguish it from other programming languages — features like generators, goal-directed evaluation, and string scanning. Like most other programming languages, Icon has some features that aren't used often but are nonetheless essential for certain tasks. Random-access input and output are examples in this category. Icon also has some features that aren't used very often and aren't essential either — excess baggage that seemed good at the time it was added.

We've had an interest for some time in analyzing Icon programs to get a better understanding of the importance and utilization of Icon's various features. At this point in Icon's development, this primarily is an exercise in hindsight, but it's nonetheless interesting and it might be helpful to other programming language designers.

There are two approaches to measuring the use of programming language features: static and dynamic. Static analysis is concerned with how often different features appear in the text of programs, such as how many instances there are of the function find(). Dynamic analysis is concerned with how often different features are used during program execution, like how often find() is called.

In this article we'll consider static analysis and leave dynamic analysis to later.

Obtaining Static Information

Static analysis deals only with the text of programs. You might imagine several ways of analyzing program text, such as writing a program that parses Icon programs and tabulates the different kinds of syntactic tokens it finds. This is a daunting task, even in a high-level language with good string-processing facilities, such as Icon. Icon's syntax is complex and extensive; producing a correct and complete parser for it is difficult and tedious, even using specialized tools like lex and yacc. Of course, a parser for Icon already exists in the implementation of Icon itself. It would be nice to be able to use it. The Icon translator generates ucode for the virtual Icon machine [1,2]. Ucode is much simpler syntactically than Icon itself, so getting static information about Icon programs from the corresponding ucode is a possibility. There's a problem with this, however; some Icon expressions, notably control structures, are translated into sequences of ucode instructions and it's by no means easy to reconstruct the original expressions in these cases.

The kind of problem we're considering here motivated the development of variant translators [3, 4], which use Icon's parser to produce various forms of text corresponding to an Icon program. At the core of the variant-translator system is an "identity" translator that translates an Icon program into

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

cs.arizona.edu (cd /icon)

a semantically equivalent one; only the layout differs between the input program and the output program. A specification system then allows the output to be changed to other forms.

Some time ago we wrote a variant translator to translate an Icon program into another program, which when executed, tabulated the tokens in the original program. (If this doesn't make a lot of sense, bear with us; we'll be more specific later.)

We struggled with this variant translator over a period of time. It was complex and stretched the capability of the variant-translator system as well as our own skill. We got this variant translator working to the point that it produced most of the information we wanted, but it never was entirely complete and correct.

When we went to use this variant translator to get information for this article, we discovered it didn't work anymore — recent changes to Icon's syntax, like preprocessor directives, were interpreted as syntactic errors. The variant translator was a victim of "progress".

We thought about trying to bring the old variant translator up to date, but we decided instead to use the recently developed meta-variant translator system [5]. We've since changed our terminology and now call these just meta-translators [6]; the concepts haven't changed. Using a meta-translator enabled us to work in Icon, instead of C.

Producing a meta-translator for the static analysis of Icon programs turned out to be a snap. We had something running in less than an hour, and a complete version — one that did much better than the original variant translator — wasn't long in coming.

A meta-translator reads an Icon program (the source program and writes a single Icon procedure, Mp(), that contains calls to procedures for every syntactic token in the source program. The procedure Mp() thus contains a complete representation of the source program.

The procedure Mp() is combined with code-generation procedures for every kind of Icon token:

```
procedure main()

   # initialization code

   Mp()

   # termination code

end

procedure Mp()

   # translation of source program

end

# code-generation procedures called by
# Mp().
            ...
```

What happens when this program is run depends on the code-generation procedure used. In an "identity" translation, these procedures just produce strings that correspond to the token they represent.

For example, Binop_(op, e2, e2) in the identity translator is:

```
Binop_(op, e1, e2)
   return cat("(, e1, " ", op, " , e2, ")"
end
```

The procedure cat() from the Icon program library produces the concatenation of an arbitrary number of strings. It makes writing multiple concatenations easier.

An expression in the source program, such as

```
i + 10
```

is translated into

```
Binop("+", Var("i"), Ilit(10))
```

in the procedure Mp(). As you might guess, Var() and Ilit() just return their arguments. Thus, when Mp() is called with the identity-translation procedures,

```
Binop("+", Var("i"), Ilit((10))
```

produces "(i + 10)", which then is written out. The parentheses are provided to avoid problems with precedence and associativity in compound expressions.

This is quite a bit of mechanism to do nothing more than translate a target program into a semantically equivalent one that is laid out somewhat differently from the target program. What the identity procedures provide is a model for other, more interesting translations. For static analysis, for example, the procedure Binop() has this form:

```
procedure Binop_(op)
   binop["e1 " || op || " e2"] +:= 1
   return
end
```

Thus, every time Binop() is called by Mp(), the entry in the table binop for that operator is incremented. For example, i + 10 in the target program increments the entry in binop for "e1 + e2". Other instances of addition operations increment the same entry. The constant strings showing "e1" and "e2" are provided so that the results are easy to understand.

There are similar tables and procedures for the other kinds of syntactic tokens. These tables are created in the main program of the meta-translator before Mp() is called, and when Mp() returns, the contents of the tables are written out. The result of this translation is not a program at all; instead it's a tabulation of the tokens in the source program.

If you're not quite sure what's going on, look at identgen.icn (for identity translation) and tokgen.icn (for token tabulation) in Version 9 of the Icon program library.

Selecting Programs for Static Analysis

With the token-tabulation meta-variant translator, all we have to do for static analysis is to find some suitable target programs and produce tabulations of their tokens.

The problem is selecting a suitable set of target programs. Although we have accumulated thousands of Icon programs written by hundreds of authors over a period of years, there's no way to know that these are programs are in any way representative of all Icon programs. In fact, the concept of "representative" is not really well-defined. Static analysis using the token-tabulation meta-variant translator is relatively fast, but compiling data from thousands of programs is time consuming and tedious even when it's all done with programs and scripts. It's also not clear that performing static analysis on thousands of programs would produce more insight than working with a smaller sample.

For this article, we decided to use the Version 9 Icon program library. We included all the program and procedure files, but not the material in the so-called packages. (The meta-translator works just as well on collections of procedures as it does for complete programs.) This sample contains 569 files amounting to about 2.5 MB. These files contain 88,486 lines, of which 47,896 contain executable code. (The rest are blank lines and lines consisting only of comments).

The Icon program library certainly is not representative of all Icon programs, whatever we choose to mean by representative. But it does include a wide range of applications and is the work of many authors with widely differing styles. Just be aware that what follows is a static analysis of one version of the Icon programming library and the results should not be interpreted too broadly.

What to Expect?

All Icon programmers probably have some preconceived ideas about which features of Icon are used most frequently. Before going on, write down your guesses as to the answers to the following questions. We'll give the answers later but don't peek.

1. What operator appears most frequently in Icon programs?
2. What control structure?
3. What keyword?
4. What function?
5. What string literal? Integer literal? Cset literal? Real literal?
6. What variable name?

The Results

Our meta-variant translator for tabulating tokens produces a great deal of information, including a listing of all the string literals in a program. Such information is not particularly useful except as it pertains to a particular program. And, because of the size of Icon's computational repertoire, it would take a book to list all the information about every expression for just the Icon program library. What follows are highlights (and some "lowlights"), along with some notes about items of particular interest. We'll present the results mostly in terms of "popularity".

Operators: There are 44,962 instances of operators in the Icon program library. The "top ten" in terms of occurrence are listed below. The percentages given are in terms of all operators.

operator	occurrences	percentage
e1 := e2	13,013	28.94
e1 . e2	5,107	11.36
e1[e2]	4,890	10.88
e1 + e2	2,578	5.73
e1 || e2	2,402	5.34

e1 – e2	1,671	3.72
∗e	1,489	3.31
\e	1,299	2.89
e1 ∗ e2	1,243	2.76
e1 & e2	1,105	2.48

There's a fine point of terminology here. An operator is a special syntactic form. There are many other *operations*. An operation is any expression that evaluates its arguments from left to right and then performs some computation that does not interfere with control backtracking. All other expressions are control structures (by definition). For example,

```
if e1 then e2 else e3
```

is a control structure, not an operation. The control expression e1 is evaluated first; depending on whether or not it succeeds, either e2 or e3 is evaluated (but not both).

Among the operations, operator syntax generally is provided for frequently used operations or when there's a familiar corresponding notation, such as is used in mathematics. (This isn't always the case: It's hard to explain why there is an operator for refreshing co-expressions.) Functions, on the other hand, are used for most of the rest of the computational repertoire. Function invocation is, of course, an *operation*.

If we look at the popularity of operations, not just operators, the picture is very different. There are 68,376 instances of operations in the Icon program library. Invocation is by far the most popular, with 20,753 occurrences, amounting to 30.35% of *all* operations. No other operation comes out in the top 10. (A good exercise is to try to list all the different kinds of operations in Icon from memory.)

Control Structures: As mentioned earlier, any expression that is not an operation is a control structure. There are 17,073 instances of control structures in the Icon program library. The ten most popular control structures are:

control structure	occurrences	percentage
e1 | e2	3,240	18.98
return e	2,540	14.88
if e1 then e2	2,417	14.15
if e1 then e2 else e3	1,514	8.87
case selectors	1,359	7.96
every e1 do e2	1,151	6.79
while e1 do e2	668	3.91
e1 ? e2	595	3.49
fail	577	3.38
every e	459	2.69

When we started to tabulate control structures, we realized we had a problem a problem — e1; e2 is a control structure (it inhibits backtracking), but the meta-translator can't detect the implicit semicolons that are provided by the Icon translator automatically where an expression ends at the end of a line and another expression starts on the next line.

Even the explicit semicolons occur more frequently than any other control structure. Since we couldn't count implicit semicolons, we decided to omit semicolons altogether from the figures above.

As an aside, you might note that the analysis of Icon programs brings out some fine points in Icon's syntax and semantics. You'll see more of this, but from a different perspective, when we publish an article on dynamic analysis.

Keywords: There are 2,153 occurrences of keywords in the Icon program library. Here are the five most popular keywords are:

keyword	occurrences	percentage
&null	323	15.00
&digits	193	8.94
&errout	180	8.36
&window	135	6.27
&pos	100	4.64

Functions: It isn't possible with static analysis to tell whether an identifier that's the name of a function has a function value when it's used. As we've pointed out before [7], it's possible to do something like

```
tab := 8
```

Such uses are relatively rare, as are uses of functions other than by their names. However, two identifiers, args and name, are the names of functions but are more commonly used as local identifiers, as in

```
procedure main(args)
```

We've deleted these two identifiers from the following tabulation of function names, since they appear frequently but each actually is used as a function only once.

There are 14,478 occurrences of function names in the Icon program library, of which 136 are distinct. (There are 139 functions in the version of Icon we used for comparison. The three functions not used in the library are chdir(), errorclear(), and loadfunc(), a function available on some UNIX platforms for dynamically loading C routines.) The 10 most popular functions are:

function	occurrences	percentage
write()	1,464	10.11
tab()	1,417	9.79
writes()	697	4.81
stop()	631	4.35
put()	627	4.33
move()	539	3.72
find()	452	3.12
upto()	430	2.97
integer()	406	2.80
many()	378	2.61

Literals: It seems a bit silly to list information about literals, but as a curiosity (or for some future trivia quiz), here are the "highlights". There are 32,469 occurrences of literals in the Icon program library, of which 7,315 are distinct.

There are 15,780 occurrences of string literals, of which 6,632 are distinct. The five most popular string literals are:

literal	occurrences	percentage
""	836	5.30
" "	485	3.07
","	143	0.91
"-"	130	0.82
"\n"	126	0.80

The longest string literal in the library has 17,285 characters. It's an image string used to draw playing cards.

There are 15,165 occurrences of integer literals, of which 319 are distinct. The five most popular integer literals are:

literal	occurrences	percentage
1	5,414	35.70
0	2,619	17.27
2	1,813	11.96
3	655	4.32
4	523	3.45

Note that there is much less "diversity" in the use of integer literals than of string literals.

There are 781 occurrences of cset literals, of which 203 are distinct. The five most popular cset literals are:

literal	occurrences	percentage
'\t '	128	16.39
' '	65	8.32
','	49	6.27
':'	28	3.59
'('	24	3.07

There are 743 occurrences of real literals, of which 161 are distinct. The five most popular real literals are:

literal	occurrences	percentage
1.0	136	18.30
0.0	101	13.59
0.5	81	10.90
2.0	27	3.63
0.25	25	3.36

There are 69,876 occurrences of variable names, excluding function names, of which 4,863 are distinct. The five most popular non-function variable names are

name	occurrences	percentage
s	2,191	3.14
i	2,085	2.98
self	1,284	1.84
x	1,153	1.65
line	952	1.36

The reason that self ranks so high is that it's used extensively in the graphics tool kit, which originally was written in Idol [1] — a reminder that the Icon program library is not representative of all Icon programs.

Declarations: Just to complete the picture, there are 7,777 declarations in all:

declarations	occurrences	percentage
procedure	4,699	60.42
local	1,825	23.47
link	432	5.55
global	331	4.26
static	300	3.86

record	171	2.20
invocable	19	0.37

In the case of scope declarations, each identifier counts as a declaration even if several follow the reserved word. The same is true of link declarations.

Unpopular Expressions: Listing the least frequently used expressions in the library isn't particularly revealing. Some operators appear only in proto.icn, a program whose sole purpose is to list all syntactic forms.

The absence of some operators isn't really surprising, since there are augmented assignment operators for all binary non-assignment operators. Can you actually imagine using

```
e1 &:= e2
```

or

```
e1 @:= e2
```

If we ignore augmented assignment operators, there are a few operators whose unpopularity surprised us:

operator	occurrences	percentage
e1 ** e2	5	0.01
e1 <– e2	37	0.08
~e	38	0.08
e1 ||| e2	40	0.09

The five least popular control structures are:

control structure	occurrences	percentage
suspend e1 do e2	3	0.02
until e	12	0.07
create e	39	0.23
|e	39	0.23
until e1 do e2	39	0.23

Earlier we saw that string scanning was only the eighth most popular control structure. String scanning expressions, however, tend to be more complicated than those for other control structures, as is indicated by the popularity of tab() and move().

Conclusions

By now, you've seen the answers to the questions that we posed earlier. To summarize:

	winner	our guess
operator	e1 := e2	e1 := e2
control structure	e1 | e2	while e1 do e2
keyword	&null	&digits
function	write()	write()
string literal	""	" "
integer literal	1	1
cset literal	'\t '	''
real literal	1.0	1.0
variable name	s	x

We figure we did pretty well, only being really off on the most popular control structure. How did you do?

What conclusions can we draw from all this? There are some things that are as should be expected, some things that point out how Icon programs should be viewed, and a few surprises (for us, at least).

The big question remains: How do the results of static analysis relate to what actually goes on when a program is run? As noted earlier, for this we need dynamic analysis, which we'll deal with in a subsequent article.

Now with some trepidation, we'll make an offer. If you have an Icon program for which you'd like a static analysis, send it to us by e-mail or on an MS-DOS or Macintosh floppy (no printed listings, please). We'll return a static analysis of your program in a like manner. One program only, please.

If you're interested in the static analysis of a file from the Icon program library, send us the file name and we'll send you back the information. Please do not ask for more than one.

Address e-mail related to this offer to

ralph@cs.arizona.edu

not to icon-project and *certainly* not to icon-group.

This is a limited-time offer, which means that if we're inundated with requests, we'll run up a white flag.

If you'd like to do your own static analysis and are running Version 9 of Icon on a UNIX platform, you can get the necessary meta-translator by anonymous FTP to cs.arizona.edu; cd /icon/meta and get the READ.ME there.

We expect to have meta-translators available for other platforms at some time in the future.

Watch the *Icon Newsletter* for an announcement.

References

1. “An Imaginary Icon Compiler”, The Icon Analyst 8, pp. 2-6.

2. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986, pp. 111-126, 264-278.

3. “Variant Translators”, The Icon Analyst 7, pp. 2-5.

4. *Variant Translators for Version 9.0 of Icon*, Ralph E. Griswold, IPD245, 1994.

5. “Meta-Variant Translators”, The Icon Analyst 23, pp. 8-10.

6. *Building Source-Code Processors for Icon Programs*, Ralph E. Griswold, IPD263, 1994.

7. “Programming Tips”, The Icon Analyst 25, pp. 11-12.

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

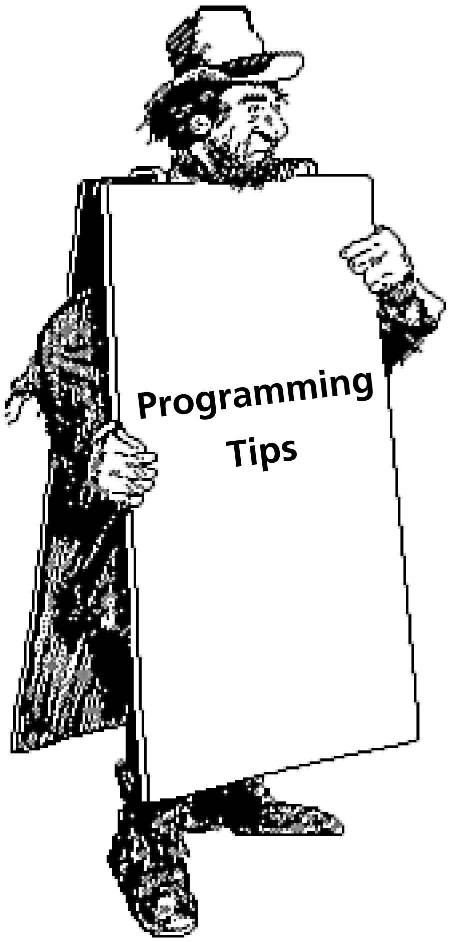
or

...uunet!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona



Preprocessor Definitions

This is the third tip in a series on avoiding bugs. Here we deal with a newer feature of Icon that some of you may not have used.

Icon’s preprocessor allows a name to be assigned to an arbitrarily complicated expression. A simple example is

```
$define SIZE      width + offset
```

When SIZE is used subsequently in the program, width + offset is substituted for it.

Suppose SIZE is used as follows:

```
dimension := SIZE * 3
```

This groups as

```
dimension := width + (offset * 3)
```

where the obvious intention was

```
dimension := (width + offset) * 3
```

The value assigned to dimension almost certainly will be incorrect and result in a bug that may be hard to find — after all

```
dimension := SIZE * 3
```

looks correct.

The solution is easy: Use parentheses in the definition, as in

```
$define SIZE    (width + offset)
```

Then

```
dimension := SIZE * 3
```

is equivalent to

```
dimension := (width + 3) * 3
```

as intended.

Before you say that the preprocessor should supply parentheses for you, recall that preprocessor definitions can be used for many things other than complete expressions. You have to take care of parentheses yourself.

Another thing to watch out for when using the preprocessor is using a name that already has a meaning in Icon. For example,

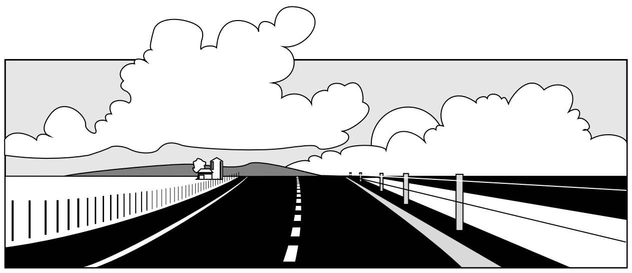
```
$define tab 8
```

results in an instance of the function call like tab(i) turning into 8(i) — not exactly conducive to correct program behavior. One way to avoid this kind of bug is to use uppercase letters for the names of defined constants.

Avoiding bugs like these, as well as the ones discussed in the previous two programming tips, involves discipline. All good programmers know that discipline is important, but they don't always practice it. The fact that Icon encourages a free-and-easy style does not help. But it can take a lot of time and effort to find bugs in a program, especially the kinds of bugs described in these recent tips. We don't know of anyone who prefers debugging to programming; following a few simple rules faithfully can make the whole programming process much more pleasant. Enough preaching; we just hope that these tips, which come from our experience, will help.

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

By now, you've probably read all you want to read about Lindenmayer systems. We promise to leave the subject after an article on compiling 0L-system specifications in the next issue.

We'll follow up the article on the static analysis of Icon programs in this issue by the first in a series of articles on the dynamic analysis of Icon programs — determining what goes on during program execution. This is an extensive subject and we expect to have several articles on it in future issues of the Analyst. We'll probably start by explaining how program execution can be monitored in Icon and what kinds of information can be obtained by this means. We'll then give some results and compare them with the results of static analysis. We probably will get into ways of presenting program activity visually, although not being able to print in color (or rather, not being able to afford to print in color) limits what we can do.

In the next issue, we also expect to have an article on string invocation, which allows Icon's functions and operators to be invoked using their string names.