The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1995
Number 30

In this issue ...

Subscription Renewal

For many of you, this is the last in your present subscription to the Analyst and you'll find a subscription renewal form in the center of this issue. Renew now so that you won't miss an issue.

Your prompt renewal also helps us in planning and by reducing the number of follow-up notices we have to send.

The Versum Problem

Recreational mathematics — mathematics for the fun of it — is a popular activity for both amateurs and professionals. Topics in recreational mathematics are wide-ranging: arithmetic, number theory, geometry, topology, combinatorics, and many subjects that don't fall into any well-defined category [1]. The immense popularity of Martin Gardner's long-running "Mathematical Games" column in *Scientific American* and his many books are testimony to how widespread interest is in recreational mathematics.

In recent years, increasingly easy access to personal computers has changed the character of recreational mathematics by making it feasible to carry out formerly impractically tedious and time-consuming calculations.

In this article, we look at an unsolved problem in number theory. Number theory may seem like a strange topic for Icon, but this problem involves string manipulation as well as arithmetic, and its solution may lie in pattern matching. Here's the problem:

> Take a positive integer, expressed in decimal form. Reverse the order of its digits from end to end, and add the result to the original number. Continue this process to see if you get palindromes — numbers that read the same from left-to-right and right-to-left. This is called the versum problem, short for the reversal sum problem.

There is a long-standing conjecture that all versum sequences contain palindromes. A stronger conjecture is that all versum sequences contain an infinite number of palindromes. The original conjecture is based on the observation that palindromes usually appear quickly. For example, starting with 168, a palindrome occurs after only three reversal-additions and others follow:

168
1029
10230
➳ 13431
➳ 26862
53724
96459
191928
1021119
10132320
➳ 12455421
24910842
49712784
...

The versum sequence itself starts with the first sum (1029 in this case), but we show the integer that starts it for clarity.

A problem was discovered quickly, however. For some numbers, a palindrome does not show up quickly. 196 is such a number:

196
887
1675
7436
13783
52514
94039
187088
1067869
10755470
18211171
35322452
60744805
111589511
227574622
454050344
897100798
1794102596
8746117567
16403234045
70446464506
130992928913
450822227944
900544455998
1800098901007
8801197801088
17602285712176
84724043932847
159547977975595
755127757721546
1400255515443103
4413700670963144
8827391431036288
17653692772973576
85191620502609247
159482241005228405
664304741147513356
1317620482294916822
3603815405135183953
7197630720180367016
13305261530450734933
...

At the time the problem was first studied, computations had to be done by hand and it was assumed that if the process was continued long enough, a palindrome would show up. More recently, using computers the versum sequences for 196 and other "intractable" integers have been run for millions of steps without producing a palindrome. Now the opinion of almost all persons who have studied the versum problem is that there are infinitely many versum sequences that do not contain palindromes. So, if you're interested in working on this problem, you're probably more likely to make headway by trying to show the original conjecture is false.

It doesn't take much to implement the computation of versum sequences. Here's a little program that takes an integer on the command lineand writes the integer and its versum sequence with asterisks after palindromes. The program runs until it is interrupted.

```
procedure main(args)
  i := (0 < integer(args[1])) |
   stop("*** positive integer not provided")
  repeat {
    writes(i)
    j := reverse(i)
    if i == j then write("*") else write()
    i +:= j
    }
end
```

This program shows the usefulness of large integers, since the integers quickly exceed the limits of native machine integers, as illustrated by the sequence for 196 . The largest native integer for 32-bit architecture is 2,147,483,647 and is exceeded by the 18th value in this sequence. The thousandth value has 411 digits; the numbers quickly get really large by ordinary standards.

Before going on, it's worth examining this program. Efficiency always is a concern in computations that run for many iterations, which certainly is the case here. It may not be obvious that such a simple program could be made to run much faster. The key to a faster program lies in recognizing that Icon converts types automatically to suit the needs of operators. In this program there are automatic conversions from integers to strings and vice versa. This makes the program easy to write — you don't have to worry about doing the conversions yourself. But in this convenience, there is a problem that may not be evident. In the loop in the preceding program i is an integer. writes(i) converts the integer to a string to write it and then reverse(i) converts it to a string again. At this point, j is a string. In the next step, i is converted to a string a third time for string comparison. (The use of integer comparison instead would convert j from a string to an integer.) Finally, j is converted to an integer when it is added to i.

Obviously, some conversions are necessary. The integer i must be converted to a string to reverse it, and j must be converted to an integer to add it to i. But the program can be rewritten to avoid the other conversions. Is this worth doing? A hint lies in the fact that conversion of a large integer to a string is quadratic in the number of digits [2]. This doesn't amount to much for "small" large integers, but it's a killer for "really large" large integers.

It doesn't take much to revise the program to avoid unnecessary conversions. One way is:

```
repeat {
    si := string(i)
    writes(si)
    sj := reverse(si)
    if si == sj then write("*") else write()
    i +:= sj
    }
```

According to our argument, the elimination of the two unnecessary conversions of i to a string should result in increasingly improved performance as the computation continues. Here are timings in seconds for the versum sequence for 196 on the two versions of the program, run on an Alpha 200 4/233:

iterations	version 1	version 2
500	1.69	0.72
1000	12.36	4.87

These figures were obtained with storage regions that were so large that no garbage collections occurred. With the default 65KB region sizes, the count of garbage collections was:

iterations	version 1	version 2
500	4	2
1000	16	10

The timings, however, were not that much different:

iterations	version 1	version 2
500	1.91	0.81
1000	13.02	5.14

The reason that the extra garbage collections didn't have much effect on the timings is that when a garbage collection occurs, all accumulated data except the current strings are collectable. Garbage collection does not even look at "garbage" (It might better be called "storage reclamation", but the term garbage collection is firmly imbedded in the jargon of computing.) It's the data that has to be saved that takes time in the garbage-collection process. This can be a very significant factor in programs that produce large in-memory databases, but that's not the case here.

While we're discussing efficiency, we should mention that some Icon programmers think it's faster or somehow "safer" to perform explicit conversions, as in

```
i +:= integer(j)
```

That's not faster; it's actually a little bit slower, since there's a function call in addition to the conversion, which has to be done in either case.

Another question concerns the comparison that is used for testing for palindromes. The loop might be cast as

```
repeat {
    si := string(i)
    writes(si)
    j := integer(reverse(si))
```

```
        if i = j then write("*") else write()
        i +:= j
        }
```

This version shows no detectable difference in running speed compared to the former one. Take your pick.

But let's get back to the versum problem. How might you approach it using programs as tools? Versum sequences might be analyzed for digit frequencies, recurring patterns, and so forth. Give the problem a try and let us know if you discover anything interesting. If you are able to prove or disprove the conjecture, you'll be famous, at least in among persons interested in recreational mathematics.

In addition to the conjectures posed above, there are other questions you might consider:

(1) Are there any versum sequences that contain an infinite number of palindromes?

(2) If not, what is the maximum number of palindromes in a versum sequence and for what starting values does it occur?

(3) Is there any correlation between the properties of an integer and the number or frequency of palindromes in its versum sequence?

(4) Are all positive integers found in some versum sequence?

(5) If not, what percentage of all integers are found in some versum sequence?

It's known that many versum sequences "converge" in the sense that at some point they produce the same value, after which all subsequent values are the same. These are called versum tails.

(6) Do all versum sequences converge to the same sequence?

(7) If not, is there a finite number of common versum tails?

(8) If there are versum sequences that do not contain palindromes, do they all have a common tail?

It's easy to ask hard, even intractable questions. While proofs may be hard to come by, much evidence can be gathered with fairly simple programs. You might try graphical representations of versum sequences in studying these questions. Let us know if you find something of interest.

We'll have some "results" of our own in the next issue of the Analyst, along with some other things that are known about versum sequences.

References

1. *A Bibliography of Recreational Mathematics: Volume3*, William L. Schaaf, National Council of Teachers of Mathematics, Reston, Virginia, 1973.

2. "Large Integers", Icon Analyst 4, pp. 5-6.

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

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: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...uunet!arizona!icon-project

THE UNIVERSITY OF
ARIZONA ®
TUCSON ARIZONA

and

Bright Forest Publishers
Tucson Arizona

From the Wizards

Sometimes initialization is required before any one of a set of procedures is used. An example is the turtle graphics package in which a window and the turtle state have to be set up before any drawing can be done.

One way to handle this kind of situation is to require the user to call an initialization procedure, TInit(), before using other procedures in the turtle graphics package:

```
procedure TInit()

   if /T_win then {
      if /&window then
         WOpen("width=500", "height=500") |
          stop("can't open window")
      T_win := &window
      }
   T_stack := []
   T_x := WAttrib(T_win, "width") / 2 + 0.5
   T_y := WAttrib(T_win, "height") / 2 + 0.5
   T_deg := –90.0

   return

end
```

An approach that is more friendly to users is to call TInit() in the initial clauses of the other procedures, as in:

```
procedure TGoto(x, y)
   initial TInit()

   T_x := x
   T_y := y

   return

end
```

If nothing else is done, the first call of a different turtle graphics procedure would call TInit() in its initial clause, with disastrous results.

The most obvious thing to do is to put a switch in TInit() to record that it has been called:

```
procedure TInit()
   static done

   if /done then done := 1 else return

   if /T_win then {
      if /&window then
         WOpen("width=500", "height=500") |
          stop("can't open window")
      T_win := &window
      }
   T_stack := []
   T_x := WAttrib(T_win, "width") / 2 + 0.5
   T_y := WAttrib(T_win, "height") / 2 + 0.5
   T_deg := –90.0

   return

end
```

A better method is to put the entire procedure body for TInit() in an initial clause:

```
procedure TInit()

   initial {
      if /T_win then {
         if /&window then
            WOpen("width=500", "height=500") |
             stop("can't open window")
         T_win := &window
         }
      T_stack := []
      T_x := WAttrib(T_win, "width") / 2 + 0.5
      T_y := WAttrib(T_win, "height") / 2 + 0.5
      T_deg := –90.0
      }

   return

end
```

With this method, TInit() simply returns on every call but the first.

But that's not clever enough for an Icon Wizard. Look at this:

```
   procedure TInit()

      TInit := 1

      if /T_win then {
         if /&window then
            WOpen("width=500", "height=500") |
             stop("can't open window")
         T_win := &window
         }
      T_stack := []
      T_x := WAttrib(T_win, "width") / 2 + 0.5
      T_y := WAttrib(T_win, "height") / 2 + 0.5
      T_deg := –90.0

      return

   end
```

Since TInit() is a procedure, TInit is a global variable. In previous articles we've mentioned the hazards of accidentally assigning a value to a variable that is the name of a procedure. But here it serves a very useful purpose. By assigning 1 to TInit, TInit() effectively wipes itself out. Any subsequent calls to TInit() are equivalent to 1(), which is faster than calling a procedure that just returns. Granted, each turtle graphics procedure only invokes TInit() once, so speed isn't the issue. The issue is the exercise of Icon wizardry.

Pity the C or Pascal programmer who reads the code.

Bogus Expressions

In past issues of the Analyst, we've mentioned programming pitfalls and expressions that simply are bogus [1]. Here are a few more bogus expressions that we've accumulated:

1. write(&error, ...)

2. |*expr*

3. every &time()

4. return := *expr*

5. every 1 to i to {
   expr
   }

The first expression presumably should use &errout, not &error. The value of &error is an integer, with is prepended to the line written to standard output.

In the second expression, the second repeated alternation simply is redundant. This expression does the same thing as

|*expr*

The third expression "calls" the value returned by &time. This value is an integer, usually greater than 1, in which case &time() simply fails without doing anything.

The fourth expression is something that recently victimized us. You might think it's syntactically incorrect, but it isn't. Icon is an expression-based language, return is an expression, just as much as

x := *expr*

is. Consequently, in evaluating the left-hand side of the assignment expression, return is evaluated, and the procedure returns before the right-hand side of the assignment is evaluated. Since return itself has no argument, the null value is returned.

In the fifth expression, the second to presumably was accidentally written instead of do. Again, you might think the expression would be syntactically erroneous, but it isn't. It parses as

every ((1 to i) to {*expr*})

The first to expression provides the left argument for the second to expresssion.The expression inside the braces provides the right argument to the second to expression. If the expression in braces doesn't produce an integer, a run-time error occurs. If it does produce an integer, nothing much happens except that *expr* is evaluated several times. We'll leave it to you to figure out how many.

Reference

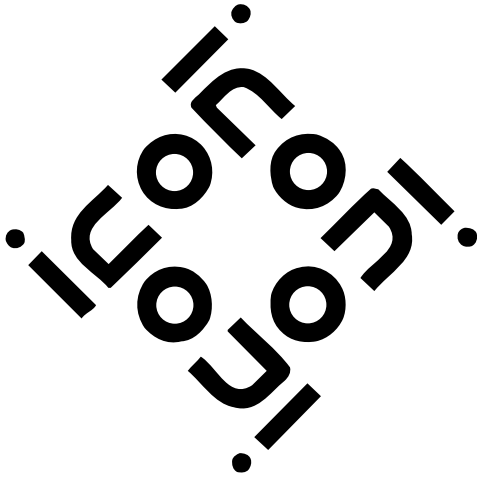
1. "Bogus Expressions", Icon Analyst 9, pp. 1-2.

Dynamic Analysis

This is the third in an ongoing series on the dynamic analysis of Icon programs — what goes on when programs execute.

Expression Activity

In the last issue of the Analyst, we showed summaries of expression activity for two of our 11 test programs, but we didn't have the space for a listing of the composite activity for all the programs. It's given on the next two pages.



name	calls	returns	suspends	failures	resumps	removals
e1[e2]	673957	673955	0	2	0	0
e1 := e2	597356	597356	0	0	0	0
e1 == e2	265588	13829	0	251759	0	0
e1 + e2	256275	256275	0	0	0	0
/e	144006	49022	0	94984	0	0
e1 – e2	143450	143450	0	0	0	0
\e	138438	44503	0	93935	0	0
writes()	135579	135579	0	0	0	0
e1 = e2	131433	49198	0	82235	0	0
?e	101646	101646	0	0	0	0
e1 === e2	91709	41484	0	50225	0	0
*e	91672	91672	0	0	0	0
find()	70809	0	5319	65491	1	5318
write()	67277	67277	0	0	0	0
e1 > e2	64183	19211	0	44972	0	0
e1 || e2	57600	57600	0	0	0	0
iand()	56867	56867	0	0	0	0
e1 <– e2	53526	0	53526	53526	53526	0
ishift()	53236	53236	0	0	0	0
ord()	48028	48028	0	0	0	0
get()	47651	46793	0	858	0	0
type()	41481	41481	0	0	0	0
tab()	39094	0	39094	7	7	39087
map()	36549	36549	0	0	0	0
e1 :=: e2	35701	35701	0	0	0	0
char()	33256	33256	0	0	0	0
right()	31940	31940	0	0	0	0
e1[e2:e3]	31174	31174	0	0	0	0
repl()	29611	29611	0	0	0	0
upto()	29445	0	24902	4544	1	24901
e1 >= e2	28858	18540	0	10318	0	0
e1 ~== e2	25521	24980	0	541	0	0
e1 < e2	24491	8616	0	15875	0	0
–e	21714	21714	0	0	0	0
many()	20983	17076	0	3907	0	0
put()	20074	20074	0	0	0	0
e1 to e2 by e3	19314	0	203830	19310	203826	0
ior()	18959	18959	0	0	0	0
e1 ||| e2	18077	18077	0	0	0	0
reads()	15122	15119	0	3	0	0
!e	14557	0	268958	11167	265568	3388
member()	14136	1356	0	12780	0	0
+e	12544	12544	0	0	0	0
move()	12121	0	11681	440	0	11681
e1 –– e2	11200	11200	0	0	0	0
e1 * e2	9865	9865	0	0	0	0
ixor()	7730	7730	0	0	0	0
any()	5470	1963	0	3507	0	0
[...]	4786	4786	0	0	0	0
left()	2983	2983	0	0	0	0
push()	2802	2802	0	0	0	0

Composite Expression Activity for All Test Programs (continued on next page)

name	calls	returns	suspends	failures	resumps	removals
center()	1925	1925	0	0	0	0
read()	1453	1450	0	3	0	0
nonterm()	874	874	0	0	0	0
string()	702	702	0	0	0	0
e1 ++ e2	562	562	0	0	0	0
pos()	437	430	0	7	0	0
=e	124	0	43	82	1	42
table()	84	84	0	0	0	0
pull()	57	50	0	7	0	0
procrec()	24	24	0	0	0	0
e1 ~= e2	23	1	0	22	0	0
e1 / e2	18	18	0	0	0	0
integer()	17	16	0	1	0	0
e1 % e2	16	16	0	0	0	0
action()	10	10	0	0	0	0
copy()	9	9	0	0	0	0
open()	8	5	0	3	0	0
list()	8	8	0	0	0	0
where()	7	7	0	0	0	0
close()	7	7	0	0	0	0
trim()	7	7	0	0	0	0
seek()	5	5	0	0	0	0
e1 <= e2	4	2	0	2	0	0
sort()	3	3	0	0	0	0
charset()	3	3	0	0	0	0
pop()	2	1	0	1	0	0
.e	2	2	0	0	0	0
set()	2	2	0	0	0	0
~e	2	2	0	0	0	0
exit()	1	0	0	0	0	0
remove()	1	1	0	0	0	0
total	3916241	3011303	607353	820514	522930	84417

Composite Expression Activity for All Test Programs (concluded)

How should one interpret such data? Recall our former admonition that our test programs aren't representative of all Icon programs. We already pointed out that iiencode.icn performs many bit operations on integers. So does another test program, press.icn. On the other hand, few programs in the entire Icon program library perform these operations at all. Consequently, the number of calls of functions like iand() are skewed. The same thing is true of reversible assignment. It's used extensively in the two test programs for the *n*-queens problem, but not elsewhere in the test programs.

The entry that surprised us was the one at the top. We had no idea that subscripting would be the most-used operation. (We guessed assignment, which is second.) If we'd been asked in advance about subscripting, we probably would have guessed it would come out somewhere in the top 25%.

The extensive use of subscripting is not the result of one or two programs; it comes out first or second in most of the test programs and in none ranks below fifth.

Note that e1[e2] includes string, list, table, and even record subscripting. Our analysis does not tell us how subscripting is divided among these types. (We can get that information using other tools, and plan to do that later.)

We also find it interesting that only 2 of the 673,957 evaluations of e1[e2] failed. But we don't know how to interpret this. (Note that subscripting a table never fails. Perhaps this gives a hint about

what to expect when subscripting is broken down by type.)

If you're puzzled by functions in the summary that you don't recognize — nonterm(), for example — these are record constructors.

We leave you to ponder the rest of the summary. Let us know if you see anything that appears unusual or particularly interesting.

Storage Allocation

Storage allocation in a programming language like Icon is a fascinating subject and one of considerable importance in many programs [1-5].

Here are summaries of allocation for the two programs we've focused on in previous articles. The types listed are internal ones, used by Icon. The remaining columns give the number of allocations, the number of bytes allocated, the average number of bytes per allocation, and the percentage of total allocation for each type.

type	allocs	bytes	aver	pct
string	9548	64988	6.80	96.54
list element	23	1204	52.34	1.78
substring tv	22	440	20.00	0.65
list	21	420	20.00	0.62
table	2	128	64.00	0.19
hash header	2	80	40.00	0.11
table-element tv	2	56	28.00	0.08
total	9620	67316	7.00	

csgen.icn

substring tv	37632	752640	20.00	97.55
string	441	18900	45.00	2.44
total	38052	771540	20.28	

iiencode.icn

If you're not familiar with Icon's implementation, some of the types listed above may seem cryptic. Lists, sets, and tables have multiple components that are differentiated internally. For example, every list has a header block (list in the summaries above) and one or more blocks for list elements (list element). The situation for sets and tables is similar. The notation tv is an abbreviation for "trapped variable", a mechanism that Icon uses to handle assignments to string and table subscripting expressions. Trapped variables are transient and almost always collected in a subsequent garbage collection. Hash headers are used for both sets and tables. Since csgen.icn uses no sets, we can tell that the hash headers are for tables, but in programs that use both sets and tables, there's no way to tell from our analysis how they are divided. If we lump the internal types into categories that correspond to source-language types, and rename hash header to set or table, the summaries become:

type	allocs	bytes	aver	pct
string	9570	65428	6.83	97.19
list	44	1624	36.90	2.41
table	4	184	46.00	0.27
set or table	2	80	40.00	0.11

csgen.icn

string	38073	771573	20.26	100.00

iiencode.icn

In these two programs, strings dominate storage allocation. (The fact that iiencode.icn allocates only strings is somewhat unusual and tells a lot about what kind of program it is.)

Finally, here are the two forms of the total allocation for all 11 test programs.

Internal types:

type	alloc	bytes	aver	pct
string	276175	6119800	22.15	41.54
substring tv	239831	4796620	20.00	32.56
list element	23870	2109856	88.38	14.32
table-element tv	23520	658560	28.00	4.47
cset	11764	470560	40.00	3.19
list	22853	457060	20.00	3.10
table element	2313	64764	28.00	0.43
record	911	22408	24.59	0.15
hash header	155	13848	89.34	0.09
co-expression	5	10000	2000.00	0.06
table	93	5952	64.00	0.04
refresh	5	1180	236.00	0.00
set element	17	340	20.00	0.00
set	2	112	56.00	0.00
file	5	100	20.00	0.00
real	2	32	16.00	0.00
total	601521	14731192	24.48	

Refresh blocks (refresh) are associated with co-expressions.

Source-language types:

type	allocs	bytes	aver	pct
string	516006	10916420	21.15	74.11
list	46723	2566916	54.93	17.42
table	25926	729276	28.12	4.95
cset	11764	470560	40.00	3.19
record	911	22408	24.59	0.15
set or table	155	13848	89.34	0.09
co-expression	10	11180	1118.00	0.07

set	19	452	23.78	0.00
file	5	100	20.00	0.00
real	2	32	16.00	0.00

In the totals for all test programs, strings still dominate the allocation, but diversity is more evident. We also can see that we didn't pick a test program that does much real (floating-point) computation, since every real computation results in allocation of a block for a real number (real). The farther we proceed with our analysis, the more we realize how unrepresentative our test set is.

Garbage Collection

Garbage collection often is as much of a concern as storage allocation.

The number of garbage collections a program performs depends to a large extent on the sizes of its storage regions. In the current implementation of Icon, there are two types of regions, one for strings and another for blocks. (In previous implementations, there was a static region for the allocation of co-expressions. Now co-expressions are managed using C's malloc() and free() functions.)

It's worth noting that trapped variables are allocated in the block region, as are all internal types except strings proper and co-expressions. There is more allocation in the block region than in the string region for the all the test programs, despite the fact that the total allocation attributable to strings amounts to about three-fourths of all allocation. It's the substring trapped variables, which are allocated in the block region, that make the difference. (Recall the dominance of subscripting in expression activity.)

Initially there is one region of each type, but more regions are created if needed. The default size for both types of regions is 65 KB. The sizes can be changed prior to program execution by setting the environment variables STRSIZE and BLKSIZE. (The two need not be the same.)

The total number of garbage collections for all 11 programs with the default 65 KB sizes is:

string region	76
block region	123

The region designation refers to the region in which allocation was attempted, but there was not enough room left. Both string and block regions are processed during a garbage collection, regardless of which one triggered the collection.

If the regions sizes are changed to 2 MB, there are many fewer garbage collections:

string region	2
block region	1

So, assuming you have enough RAM, it's worth setting the region sizes higher. Or is it? That depends on how much time is spent in garbage collection and that, in turn, depends to a large extent on the way a program uses data.

Here are the figures on the total amount of execution time on a Sparc workstation for all 11 programs:

65 KB regions:	371.998 seconds
2 MB regions:	372.149 seconds

Do not infer from these figures that the programs actually ran a trifle slower with 2 MB regions; timings vary as much as 10% from run to run even on a lightly loaded system. But, apparently, the region sizes made little difference. As mentioned in the first article in this issue of the Analyst, large region sizes *do* improve execution speed for programs that keep a lot of data in memory, such as databases. None of our test programs has this characteristic; another indictment of our choices.

A Monitoring Program

So far, we've shown the results of dynamic analysis, but we've not shown how we got them.

In an earlier article in the Analyst [6], we showed the general form of monitoring programs and a few simple examples. The monitoring programs for getting information about expression activity and storage allocation are not much more complicated than those. In fact, the bulk of the code in such programs involves the formatting of the output.

The program on the next page is the one we

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/www/

```
link evinit
link evnames
link numbers
link options

$include "evdefs.icn"

global highlights, alloccnt, alloctot, collections, output

$define  Col1    18
$define  Col2    13

procedure main(args)
   local i, cnttotal, tottotal, cnt, tot, totalcoll, opts, item

   opts := options(args, "o:")
   output := open(\opts["o"], "w") | &output

   EvInit(args) | stop("*** cannot load program")      # initialize interface

   alloccnt := table(0)                                # count of allocations
   alloctot := table(0)                                # total allocation

   while EvGet(AllocMask) do  {
      alloccnt[&eventcode] +:= 1
      alloctot[&eventcode] +:= &eventvalue
      }

   write(output, "\n",                                 # write column headings
      left("type",Col1),
      right("number",Col2),
      right("bytes",Col2),
      right("average",Col2),
      right("% bytes",Col2), "\n"
      )

   alloccnt := sort(alloccnt, 4)                       # get the data

   cnttotal := tottotal :=0

   every tottotal +:= !alloctot

   while cnt := pull(alloccnt) do {
      cnttotal +:= cnt
      item := pull(alloccnt)
      tot := alloctot[item]
      write(output,                                    # write the data
         left(abname(item), Col1),
         right(cnt, Col2),
         right(tot, Col2),
         fix(tot, cnt, Col2, 2),
         fix(100.0 * tot, tottotal, Col2, 2)
         )
      }

   write(output, "\n",                                 # write totals
      left("total:", Col1),
      right(cnttotal, Col2),
      right(tottotal, Col2),
      fix(tottotal, cnttotal, Col2, 2)
      )

end

procedure abname(code)                                 # abbreviate event name
   local result

   result := evnames(code)

   result ?:= tab(find(" allocation"))
   result ?:= {
      tab(find("trapped variable")) || "tv"
      }

   return result

end
```

Monitor for Storage Allocation

used for getting information about allocation.

The **-o** command line option is used so that the output of the monitoring program can be written to a specified file and not mixed with the output of the program being monitored. Two tables are used to keep track of the number of allocations and the total allocation, both by allocation type. The procedure **abname()** converts the names associated with events, produced by **evnames()**, and abbreviates them to improve the appearance of the output. The output shown earlier was modified slightly to fit the constraints of the Analyst's layout.

What Else?

We've barely touched the possibilities for dynamic analysis. We have results for string construction, numerical computation, structure use, string scanning, and the Icon virtual machine itself [5]. In addition, there are many ways of looking at the data that we've not tried yet, including software visualization. We'll probably all get tired of the subject before we run out of material.

References

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

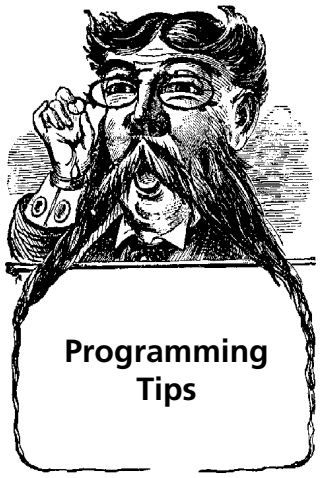
2. "Memory Monitoring", Icon Analyst 2, pp. 5-9.

3. "Memory Utilization", Icon Analyst 4 pp. 7-10.

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

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

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



Reversible Assignment

Reversible assignment rarely is used in Icon programs. In the Icon language book, it is described in the context of string scanning and in a program that produces solutions of the 8-queens problem.

There are more "everyday" situations in which reversible assignment can be useful. The basic idea is to use reversible assignment to make a tentative assignment to a variable, subject to the success of some subsequent computation. (That's the situation in the examples of reversible assignment in string scanning in the Icon language book, but the context in the examples makes reversible assignment appear to be something special for string scanning.)

A more general kind of use occurs in the common situation where a value assigned to a variable must satisfy certain conditions. For example, the value assigned to a variable that represents a size may only be valid if it's a positive integer. In this case, a check needs to be made to prevent assignment of an invalid value, perhaps provided interactively by a user.

Consider a situation where the size of a rectangle (perhaps a window) is being set and the size is changed only if both the width and height specifications are valid:

```
temp1 := getwidth()
temp2 := getheight()
if valid(temp1) & valid(temp2) then {
   width := temp1
   height := temp2
   }
else error()
```

By using reversible assignment, the temporary variables and separate assignments can be avoided:

```
(width <– valid(getwidth()) &
   height <– valid(getheight())) | error()
```

If the value produced by getwidth() is not valid, the assignment and the compound expression fail. If, however, the value produced by getwidth() is valid, it is assigned to width. If the value produced by getheight() is not valid, no assignment is made to height, but backtracking causes the value previously assigned to width to be restored to what it was originally. The second assignment could be regular assignment, since no value is assigned to height if valid() fails, but we left it as reversible assignment for generality.

For the case of testing for a nonnegative integer value, the assignment expression can be written without the use of a separate procedure, as in:

```
width <– (0 < integer(getwidth()))
```

This expression makes use of the fact that a comparison operation produces the value of its right operand.

In the more general formulation, you can also use:

```
valid(width <– getwidth())
```

If valid() fails, its argument is resumed, which reverses the assignment to width.

What's Coming Up

We have more articles on dynamic analysis in the works. One of them probably will appear in the next issue. We'll also have another article on the versum problem.

With the next issue, we'll start a series of articles on building visual interfaces for Icon programs.