The Icon Analyst

In-depth Coverage of the Icon Programming Language

October 1990
Number 2

In this issue …

The Fundamentals of Expression Evaluation

This is the second in a series of articles designed to help persons who are just getting started with Icon. Like the first article ("Getting Started"), it assumes you have prior experience in a programming language such as Pascal.

Expression evaluation is the heart and soul of Icon. While Icon has many useful features that are not found in most other programming languages, expression evaluation is more central and pervasive than any other aspect of Icon. And expression evaluation in Icon really is different from expression evaluation in most other programming languages. It's important to understand the differences and what's really going on when an Icon program runs. If you understand this, you can use the power of expression evaluation in Icon to write concise, powerful, and even elegant solutions to problems that are lengthy, tedious, and awkward to program in most other programming languages. If you don't really understand expression evaluation in Icon, you're likely to write programs that don't do what you want and that have bugs that are hard to find. It's worth making an investment in learning of few fundamentals of expression evaluation in Icon.

Ironically, the better you know another programming language, the harder it's likely to be for you to understand expression evaluation in Icon. Start with a willingness to give up familiar concepts and, more importantly, be willing to learn to think in Icon rather than trying to translate what you see in Icon into an inappropriate framework of another programming language.

The real difference between expression evaluation in Icon and that in most other programming languages has to do with how many values an expression can produce. In most programming languages, the evaluation of an expression always produces exactly one value. You may be so familiar with this concept that you'll argue that it's both natural and "the only way to do it". While many expressions in Icon do produce a single value, others may not produce any value at all or may produce many values.

Goals

What does it mean for the evaluation of an expression not to produce a value? It's not that the expression has some interesting side effect and no specific value is needed. In fact, the intent (or "goal") in the evaluation of an expression is to produce a value. For example, the goal of the evaluation of `i + j` is to produce the sum of two numbers. This goal always can be achieved, provided `i` and `j` are numbers. In Icon parlance, such an expression *succeeds*. Similarly, the goal of evaluating `read()` is to read a file and to produce the next line of input. This goal, however, cannot always be achieved. In particular, if the end of the input file has been reached, there is no data to read.

Different programming languages have different ways of handling computations that cannot be performed. Icon's way is not to produce a value. The goal is not achieved, and the expression *fails*. If that were all Icon did in such a situation, it would be very hard to write programs. Failure to meet a goal is, however, sensed by control structures. The net effect is similar to the use of Boolean values in other programming languages, but the underlying concept is much different.

An example of sensing failure is

```
if line := read() then process(line)
else write("*** eof ***")
```

This program segment attempts to read a line. If the attempt is successful, the line is processed. If, however, the attempt fails, a diagnostic message is written. It's the success or failure of `read()` that this control structure uses to select an expression to the evaluated subsequently.

Persons who have never heard of Boolean values generally find the concepts of success and failure and their use in Icon to be natural. Persons who are accustomed to Boolean values or something similar often try to interpret failure as some kind of special value that control structures intercept and interpret. This is not how Icon works — failure isn't a value and control structures can tell if an expression produces a value or not and act accordingly.

We've glossed over something in the example above that needs attention. It's the expression

```
line := read()
```

in the control expression that "drives" the if-then-else control structure. What happens to the assignment if read() fails? There's a simple explanation that applies to all situations of this kind. In order for the assignment to be performed (assignment is an expression with a goal also), it must have a value to assign. If there's no value, the assignment cannot be performed and it fails also. In this case, the value of line is not changed, since no assignment is performed. No matter how complex an expression is, if a value is needed to satisfy a goal somewhere within it and there isn't a value (because of the failure of some sub-expression), the whole expression fails. It's worth noting that in

```
if line := read() then process(line)
else write("*** eof ***")
```

if read() fails, the control structure selects the expression in the else clause and evaluation continues. The failure is isolated in the control expression and once it's processed, evaluation continues elsewhere (with another goal).

Failure and Errors

It's important to understand that the failure of an expression is not an error — it's merely a way of handling a computation that cannot be performed. In fact, many expressions in Icon are designed so that failure can be used in a natural way to control computations. For example, an out-of-bounds list subscript causes failure. This allows loops to be written without the need to know the range of allowable subscripts.

There are, of course, situations in expression evaluation that really do indicate errors. An example is the attempt to perform arithmetic on a value that is not a number. In this case, program execution terminates with a run-time error and produces descriptive information about the nature and location of the error.

The difference between failure and an error is largely a matter of language design. In Icon, failure is used for computations that are well-formed but cannot be performed, while error termination is used for situations that probably indicate mistakes or program malfunction. There is no clear dividing line between these two categories and there are several cases in which the design choice was essentially a matter of flipping a coin. Most cases are clear-cut; once you get used to the underlying differences between failure and error termination, you shouldn't have any problem with the difference.

Alternatives

Goals and success and failure are enough for simple computations. In more complex computations, especially those that involve trying various possible computations, it's helpful to have an easy way to specify alternatives.

Consider a situation in which an action is to be taken if a counter is either 0 or 1. This can be phrased as

```
if (counter = 0) | (counter = 1) then action()
```

This formulation, which looks like the logical *or* in other programming langauges, works well enough, but it's awkward and verbose — it's a difficult way to express a simple concept: "either 0 or 1". In Icon, this can be written as

```
if counter = (0 | 1) then action()
```

You may wonder how this works — what is really going on? The idea is that if one alternative doesn't lead to success, the other alternative is tried. It's important to understand more — *how* this is done.

The expression `0 | 1` has two alternatives. It produces the one on the left first. Suppose the value of counter is 1. The comparison with the first alternative (`1 = 0`) fails. Since the goal of producing a value is not met, the second alternative is produced and the comparison (`1 = 1`) succeeds; the goal is satisfied.

Alternation is a *generator* — an expression that can produce more than one value. The values of a generator are produced one at a time as they are needed. The comparison expression

```
counter = (0 | 1)
```

needs a value to perform any comparison at all, so alternation produces 0. The resulting comparison fails, so the comparison expression still needs a value. Alternation produces another value. In the case here, the comparison succeeds and no more values are needed. If the value of counter is 2, the second alternative does not meet the goal, and since there are no more alternatives, the entire expression fails.

Note that a generator produces its values one at a time, as they are needed. Only as many values are produced as are needed to achieve success of the entire expression.

A more dynamic view may help you to understand this process. When a generator produces a value, it suspends, becoming inactive but available if needed. A suspended generator is resumed if another value is needed from it. This process continues until no more values are needed or the generator has no more values (and fails). In either case, the suspended generator then "goes away".

More to Come

A closing word: Once you understand what's going on with expression evaluation, you won't have to think through every case; using it to your advantage will come naturally.

In the next article in this series, we'll describe other kinds of generators and show you how to build your own.

Syntactic Pitfalls

Every programming language has syntactic characteristics that cause problems and lead to programming errors. Icon is no exception. Knowing the most likely problems are can help you avoid them or suggest what to look for if a problem does arise.

Mistakes that are detected as syntactic errors by Icon's translator are not the real problem. These errors can be annoying and sometime hard to pinpoint (as in the case of unbalanced parentheses and braces or unclosed quotes). The real problems come from constructions that are not syntactically wrong but that produce results different from those intended.

Incorrect Operator Symbols

One of the most common errors is the use of infix = (numerical comparison) where := (assignment) is intended. It's easy to confuse these two operators because they look so much alike. The main problem is that other programming languages, such as SNOBOL4 and C, use = for assignment. (One of the most common errors in C is to use = rather than == for comparison.)

Since the infix operators = and := are in the same syntactic class, writing something like

```
count = 1
```

where

```
count := 1
```

is intended is never *syntactically* erroneous.

If you make such a mistake and are lucky, you'll get a run-time error when the expression is evaluated (perhaps because no previous value has been assigned to count, so that its value is null and hence erroneous in numerical comparison). If you're unlucky and the value of count happens to be numeric, there's no error, but count probably doesn't have its intended value afterwards.

It's so easy to overlook this kind of a mistake that programmers often don't see it even when they are looking right at it. This is all the more reason to mark this potential problem high on your list of things to look for when a program fails to run properly.

A slightly more subtle problem is the following keyboarding mistake

```
count ;= 1
```

where

```
count := 1
```

is intended. This can happen just because : and ; are upper- and lowercase on the same key on most keyboards.

You might think this mistake is a syntactic error that the translator should catch. Actually, it's perfectly legal. Worse, it executes without an error either. The semicolon separates two expressions, count and = 1 (blanks are allowed between a prefix operator and its operand). The latter expression is a string scanning operation to match 1 (actually, "1"). Since there's always a scanning subject and position, there's nothing illegal about trying to match "1"; it probably just fails silently.

Fortunately this error is easy to spot — it's not something you expect to see used intentionally.

Lines and Semicolons

There's a related problem with semicolons, discussed in the last issue of the Analyst, that is worth mentioning again.

Icon automatically inserts a semicolon at the end of a line, provided the line ends an expression and the next line begins another expression. This is a very handy feature; it allows you to write programs without having to worry about semicolons to separate expressions. (The incorrect use of semicolons is a common cause of syntactic errors in programming languages like C.) In fact, you never need to use semicolons to separate expressions in Icon. Just placing the expressions on separate lines does the trick — and improves program readability at the same time.

However, there's a catch. Because some symbols are used for both infix and prefix operators in Icon, an expression may end on one line and another expression may begin on the next line in a way that you may not anticipate. In such cases, Icon's translator cheerfully inserts a semicolon, but the result may not be what you expect.

An example of this problem is

```
s := s1
  || s2
```

The difficulty is that | is a valid unary operator (repeated alternation), so an expression ends on the first line and another expression begins on the next line. The Icon translator interprets this situation as

```
s := s1;
  || s2
```

This amounts to an assignment followed by double repeated alternation. It is syntactically correct, but semantically silly. Repeated alternation applied to an identifier just produces the corresponding variable and nothing more (since there is nothing to cause the repeated alternation to be resumed for another result). Doubling it does no more. Of course, the overall effect can be mysterious.

Applying one simple rule prevents such problems: When continuing an infix expression from one line to the next, put the infix operator at the end of the first line, as in

```
s := s1 ||
  s2
```

Now the expression does not end on the first line and no

semicolon is inserted by the translator.

Grouping Expressions

Icon has more operators than most programming languages. It uses rules of precedence and associativity to group the components of compound expressions unless the grouping is given explicitly by parentheses. For example,

```
i + j * k ^ m
```

groups as

```
i + (j * (k ^ m))
```

Icon's grouping rules for arithmetic operators are the usual ones, and you shouldn't have to worry about how arithmetic expressions group — what seems natural should work as expected (although you might be careful of exponentiation, which isn't used as often as other arithmetic operations).

The problems arise with operations like conjunction, alternation, and string scanning.

Conjunction is easy in principle. It has the lowest precedence of all the infix operations. Remembering that can save you a lot of trouble. For example,

```
i = j & j > k & m ~= n
```

groups as

```
(i = j) & (j > k) & (m ~= n)
```

Using the parentheses is a good idea here, even though they are not necessary. They make the intent of the expression clearer and easier to read.

If you're combining operations of fundamentally different kinds, you'll probably use parentheses automatically. For example, not many Icon programmers know how the following expression groups

```
i + j || "x"
```

Sometimes a grouping seems obvious but doesn't work out as expected. And example is

```
if i = 1 | 2 then expr
```

The control expression doesn't make much sense unless it groups as

```
i = (1 | 2)
```

but that's not what happens. Instead, the grouping is

```
(i = 1) | 2
```

which always succeeds!

You might fault Icon's precedence rules on this, but if you study the matter carefully, you'll find that changing the precedence of alternation so that this example groups as you'd like only raises problems in other expressions. Even if it didn't, the rules are what they are and you can't change them.

Another common grouping problem is illustrated by the following string scanning expression:

```
text ? tab(upto('.')) & pos(–1)
```

What's intended is obvious — a test to see if text ends in a period. Remember, however, that conjunction has the lowest precedence of all infix operators. The expression therefore groups as

```
(text ? tab(upto('.'))) & pos(–1)
```

Consequently, pos(–1) is not evaluated as part of the scanning expression on text but rather in the enclosing scanning expression, whatever it happens to be. Most likely pos(–1) will fail, but in any event it has no bearing on the last character of text.

The moral in all this should be clear — use parentheses to group the components of compound expressions, especially in the cases mentioned here. Parentheses may be a little extra work, but they can save you a lot of grief — and also make your program easier to read.

The Case of the Missing do

One of the most common and most annoying syntactic pitfalls comes from a "missing do". It's fairly common to see something like this:

```
while line := read()
  write(line)
```

where

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

is intended.

Since the do clause for while (as well as for until and every) is optional, the first program segment above is syntactically correct. What happens is that the entire input file is read by the loop on the first line and the second line writes only the last line that was read (or perhaps something else, if there are no lines to read).

This problem seems so difficult to find that novice programmers sometimes are certain there's something wrong with Icon itself. If you suspect that Icon is buggy when your program doesn't run correctly, you probably should think again. It's not that Icon is totally free of bugs; it certainly has some. Icon has been around a long time, however, and it is used by thousands of programmers. Any fundamental problems in its implementation were found and fixed a long time ago.

There's a form of the "missing do" problem that's particularly amusing (unless it happens to you, in which case it may be no fun at all). Consider

```
every i := 1 to 1000 {
  write(factorial(i))
  }
```

Forget the implication that you're trying to compute the factorial of positive numbers up to 1,000. You can do it with Version 8 of Icon (provided your system supports large-integer arithmetic) if you're willing to wait long enough. That's not the point, though.

The mistake here is the missing do, of course. What happens is not so obvious. This program segment does not, for example, assign 1 through 1,000 to i and then write the factorial of 1,000, as you might expect from the previous example. It doesn't write anything and just goes on to the rest of the program.

The subtlety is that second operand of to is not 1000, but 1000 applied to an argument list of a programmer-defined control operation:

```
1000 { write(factorial(i)) }
```

which is equivalent to

```
1000 ([create write(factorial(i)) ])
```

This silly-looking expression causes an attempt to select the 1,000th argument from a list of one. This fails, of course, so the to expression fails and the every loop does nothing. Maybe it's not so hard to understand why someone who experiences this problem might think Icon has gone "out to lunch".

If the braces had been omitted in the example above (they are superfluous), then just the factorial of 1,000 would have been written. On the other hand, if there had been more than one expression within the braces, as in

```
every i := 1 to 1000 {
  write(i)
  write(factorial(i))
  }
```

the translator would have detected a syntactic error.

Just so that we end on a correct note, the example above should be written as

```
every i := 1 to 1000 do {
  write(i)
  write(factorial(i))
  }
```

In some sense, dwelling on these subtleties amounts to fear mongering. If you're just beginning with Icon, don't get the impression that everything you write may lead to unfathomable bugs. Icon is really no worse than most programming languages with regard to syntactic pitfalls, and it's better than many. (Probably the worst programming language ever designed in this regard was the first version of SNOBOL, in which nothing was syntactically erroneous.) It's just that knowing the pitfalls in Icon can help you avoid them and help you know where to start looking if you encounter one.

Summary

The syntactic pitfalls listed above are by no means all of the ones you might encounter. The second edition of the Icon language book has a section at the end of each chapter on language features that lists possible syntactic problems. The problems mentioned here are just those that occur most frequently and cause the most trouble.

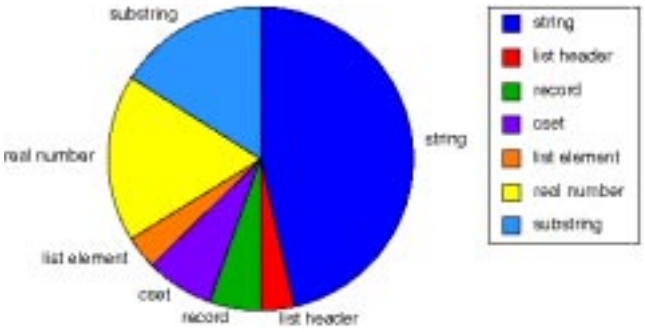
In summary,

- Watch for = where you mean := .
- If you're continuing an infix expression from one line to another, put the infix operator on the end of the first line.
- Use parentheses to group components of a compound expression the way you want them grouped.
- Watch for missing dos.

Memory Monitoring

In the last issue of the Analyst, we discussed the basic concepts of memory monitoring, described the production of allocation history files, and showed some summaries of allocation.

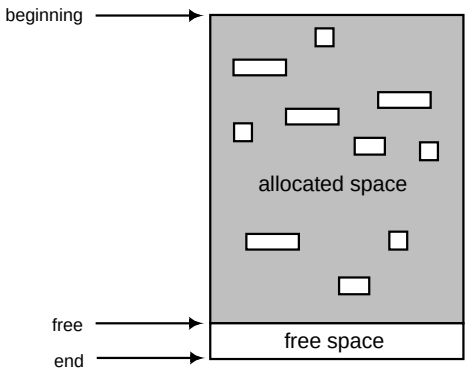
As mentioned there, the program memsum.icn in Version 8 of the Icon program library can produce output in standard tab-separated form for use in spreadsheets and charting programs. Here's a typical "pie chart":



Since an allocation history file contains all the details of storage allocation and garbage collection that occur during the course of program execution, there are many more possibilities beyond just producing summary reports and charts. An allocation history file can be "played back". There are numerous possibilities. What kinds of "instruments" could be used to play back allocation histories?

One possibility is a pictorial representation of the allocation process. There's a question of how to represent the memory in which allocation takes place. Memory can be thought of as a sequence of cells associated with increasing integer addresses. Since the memory region in which Icon allocates data is large, it's not practical to show it as one long ribbon. The usual way to handle this representation problem is to present memory as a rectangular area. The sketches used in the last Analyst to give a general idea of the allocation and

garbage collection processes in Icon are typical of this approach:



Of course the dimensions of the rectangle have no particular significance — the aspect ratio chosen is mainly a matter of typographical layout, provided it is within reasonable bounds for visual comprehension.

Visualization Programs

This same approach, but with much more detail, is used by a family of programs that produce pictures of memory based on allocation history files. These *visualization* programs come in several flavors, depending primarily on the capability of the devices that produce the pictures.

Some of these visualization programs produce "snapshots" of memory at significant points in program execution. In these snapshots, each allocated object is shown to scale according to its size and in its relative position in memory. Snapshot formats range from PostScript to bit-mapped images. Color is used, for devices that support it, to distinguish Icon objects of different types.

For devices that have the capability, the allocation process is animated, with each object appearing on the display as it is allocated. The garbage collection process is shown in a similar fashion. The effect is a "motion picture" of storage management.

Memory Displays

We can't provide animation in a printed publication like this, and we have no videos (yet).

A typical memory display is shown below. At the top is a two-line legend. The first line shows the program state at the left, the name of the allocation history file in the center, and storage information at the right. The storage information shows the region sizes first with separating plus signs. The four values in parentheses are the number of garbage collec-

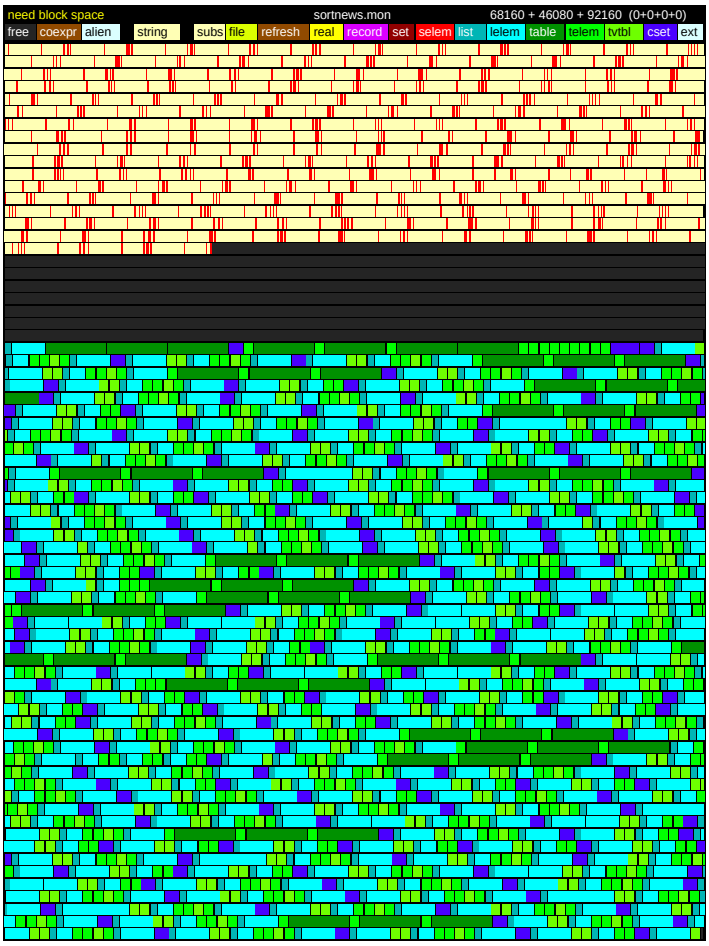


Figure 1. Memory Prior to a Garbage Collection

tions triggered by allocation in each of these regions, followed by the number of garbage collections caused by explicit calls of the function collect().

The second line of the legend lists the names of all the allocated types. Some are abbreviated because of the limited space available. Each type has a different color.

Icon's allocated storage regions follow the legend. The static region, which usually isn't very interesting, is not shown. The string region is followed by the block region. These regions are shown as being contiguous. They are contiguous in some implementations but not in others. Consequently, you shouldn't interpret the contiguity of these two regions in displays as being significant.

The allocated part of the string region shows strings in white with dark bars marking their ends. The unallocated portion of the string region is darker.

The string region in Figure 1 is fully allocated and a garbage collection is about to take place. While the types of the blocks cannot be determined easily in a black-and-white display, they are easy to identify in a color display. Look at the color picture included with this issue of the Analyst (it's from a different program than the one shown here).

The time when a garbage collection is about to take place is one of the significant times in storage management at which snapshots are taken. As described in the previous issue of the Analyst, the first phase of garbage collection "marks" all space that needs to be saved. The completion of marking is another significant point in storage management and is shown in Figure 2. The strings and blocks that need to be saved are shown in black. Notice that most of the strings that need to be saved are near the beginning of the string region. The unmarked strings represent transient allocation and storage throughput in the string region.

Figure 3 shows the same storage configuration with the coloring reversed — the space to be collected is now shown in black.

Finally, Figure 4 shows memory after the saved space has been compacted to the beginnings of the regions. The space in the string region is shown as a single string. Since overlapping substrings may be created after strings are allocated, the identification of the ends of strings after garbage collection is somewhat problematical and is not shown.

Notice that in the legend, a garbage collection is now attributed to the need for space in the string region. The block region was collected too, but not "charged" for it. Observe that garbage collection has freed a considerable amount of string space for subsequent allocation.

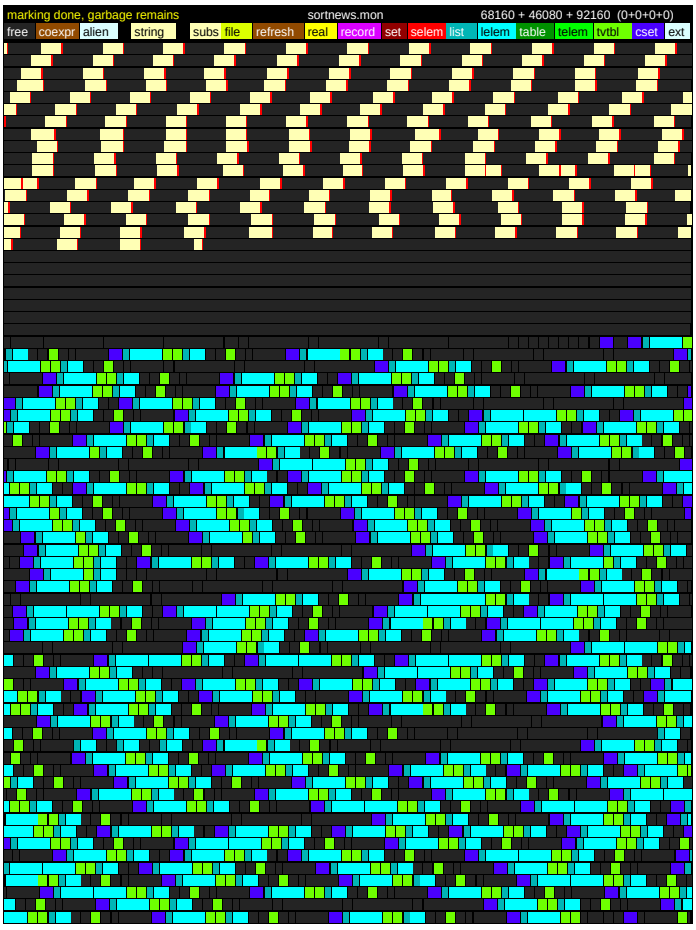


Figure 2. Memory after Marking

Uses of Visualization

There are many uses for the visualization of storage management in Icon. One use is simply to aid in understanding storage management. Another use is to get a feeling for the amount and kind of storage a program allocates as well as how much storage is transient and how much is retained.

It's sometimes possible to spot unnecessary storage allocation, especially when viewing color displays. A large number of allocated csets often indicates a program is performing unnecessary type conversion. For example, a function that expects a cset argument but gets a string converts the string to a cset. This requires allocation. Another cause of unnecessary cset allocation is the construction of the same cset over and over in a loop, when it could be constructed once outside the loop.

Memory displays also can tell the implementors of Icon a lot about how storage management performs. On more than one occasion, observation of memory displays has led to changes in the implementation.

One problem that causes poor performance in some Icon programs is "thrashing", in which a garbage collection frees very little space, so that another garbage collection is needed soon thereafter. This problem is particularly severe in programs that retain a large number of objects, all of which have to be marked on every garbage collection, but none of which is ever collected. Ways to alleviate this problem are currently under study.

Don't overlook the visual appeal of memory displays, independent of what they represent in terms of program activity. Some programs are truly beautiful in this sense. Memory displays have even inspired artists.

Version 8 of Icon includes a program for producing color PostScript snapshots of storage management. Unfortunately, this program requires too much memory for most personal computers and, of course, not everyone has a color PostScript printer or even a black-and-white one.

The source code for Icon includes most of the components needed for building visualization programs. If you have a color display, you may want to craft your own visualization program.

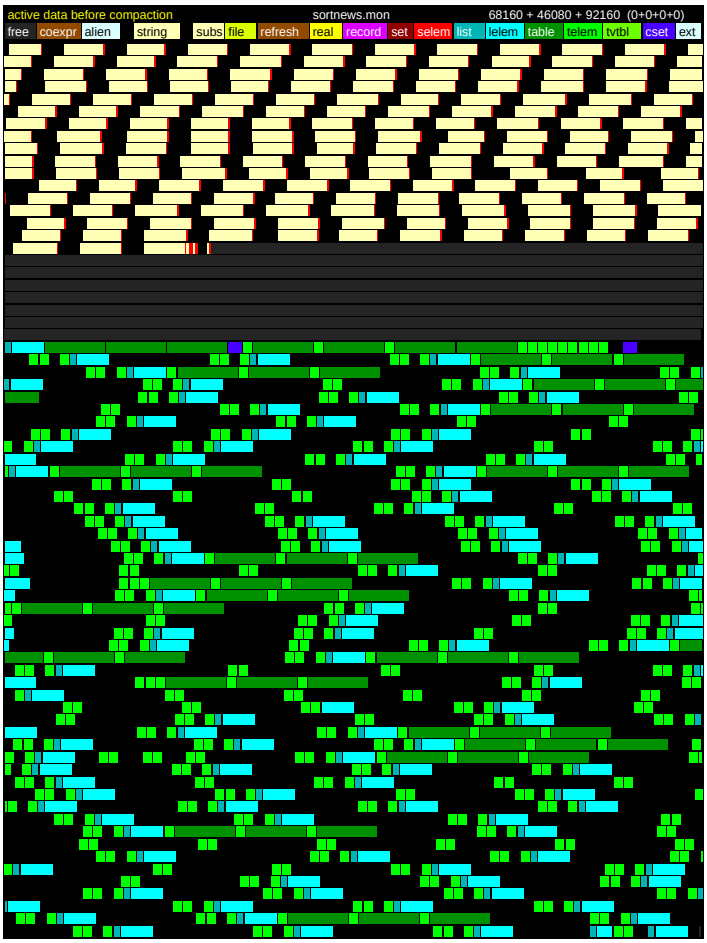


Figure 3. Memory Showing Space to be Saved

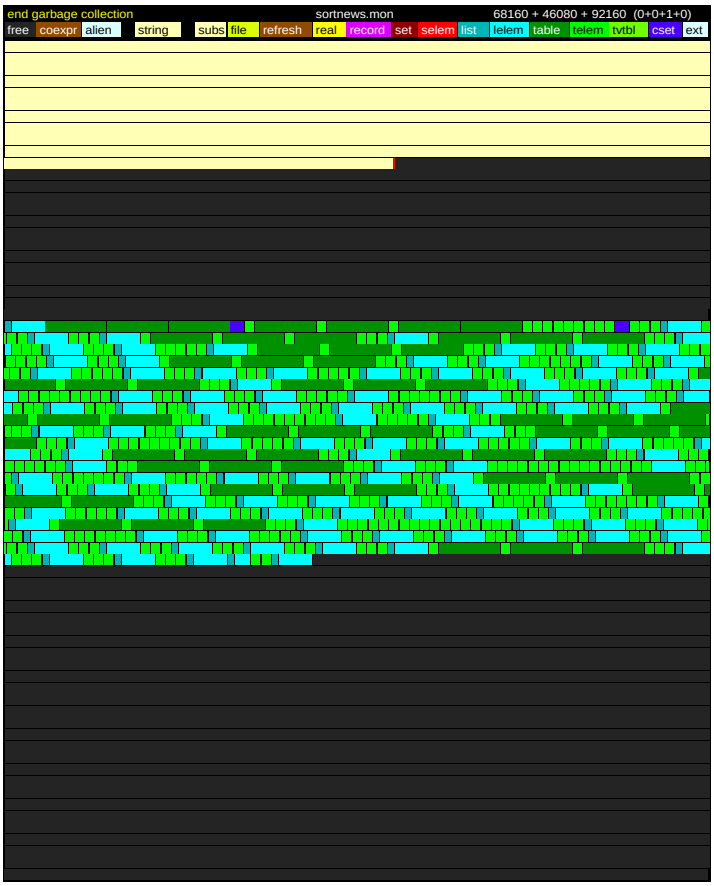
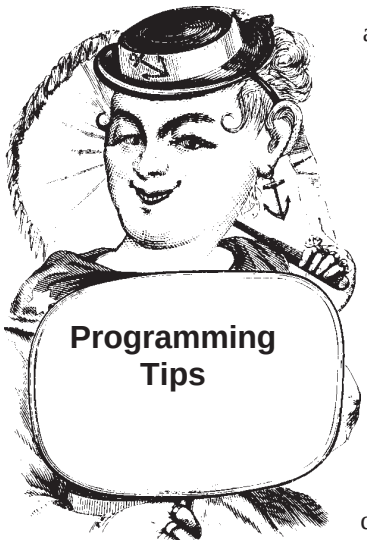


Figure 4. Memory after Compaction



Most of Icon's string analysis facilities, such as upto(c), are designed to work from left to right. If you want the last (rightmost) occurrence of a character or substring in a string, the best way to do this may not be clear.

One approach is to reverse the string and find the first occurrence (which may require looking for the reverse of something). Depending on what you want to do with the result, reversal may be unnecessarily awkward and expensive.

A different approach is to force a string-analysis function to produce all its results and use only the last one. Consider, for example, decomposing a UNIX-style path specification into its directory and file name components. An example of such a specification is:

```
path := "/usr/icon/lib/options.icn"
```

Here "/usr/icon/lib" is the directory and "options.icn" is the file name.

Here's a way to get these two pieces:

```
i := 0
path ? {
   every i := upto('/')
   if i > 0 then {
      directory := tab(i)          # up to last slash
      move(1)                      # skip the slash
      }
   else directory := ""            # no directory
   file := tab(0)
   }
```

Setting i to 0 initially allows handling of the case in which there is no directory component.

Note that it won't do to try something like

```
path ? {
   every directory := tab(upto('/'))
      .
      .
      .
   }
```

since every causes tab() to be resumed, resetting the position in the subject to 1 after the last value produced by upto().

Benchmarking Icon Expressions

In the last issue of the Analyst, we discussed timing expression evaluation. We suggested timing three expressions that produce the same results to see which was the fastest. Here are the times from a VAX 8650:

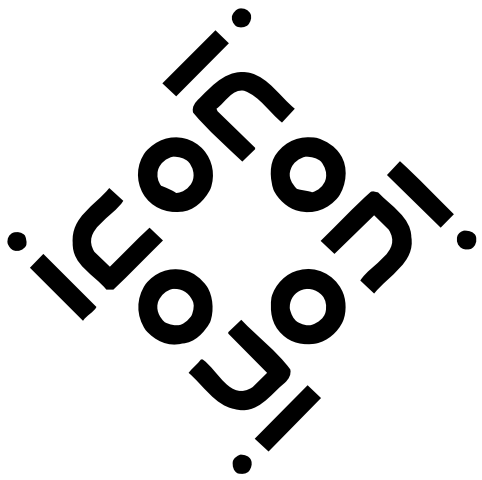
every !&digits	0.480 ms
every !"0123456789"	0.380 ms
every "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"	0.296 ms

If you think about these timing figures, you may see the reason for the comparative slowness of the first expression: &digits is a cset, but the results of !&digits are strings. Somewhere along the line, there must be type conversion. The reason why the second expression is slower than the third is less obvious, but it suggests that there is little overhead for alternation. On the other hand, the difference in efficiency may not be enough to offset the extra work in keyboarding the third expression.

In our previous discussion of timing, we looked the other way about an important issue in benchmarking expressions: storage allocation. Storage allocation is fast in Icon and, of course, the time it takes is included in the benchmark figures. The problem comes when (if) a garbage collection occurs. Garbage collection is a fairly time-consuming process. How can this be taken into account when benchmarking expressions?

There's no good answer to this. One thing to do is to try to avoid garbage collection when benchmarking, since garbage collection may distort timings significantly. If a garbage collection happens to occur when benchmarking one expression but not another, the conclusions drawn may be very misleading.

It might seem reasonable to associate a certain garbage-collection penalty in proportion to the amount of space allocated by an expression. There are several problems with this view. One problem is that many programs run to completion without ever having to do a garbage collection. In this case, the piper does not have to be paid, as it were. More perplexing is that the time it takes to perform a garbage collection depends on many factors, most of which have little to do with the amount of space allocated by a particular expression. For example, garbage collection is fastest when most of the space that has been allocated is "garbage" and does not need to be saved. Conversely, garbage collection is the slowest when there's a lot of allocated space that has to be saved (for example, for large sets and tables). In any event, there's little correlation between the space an expression allocates and how much time it takes to perform a garbage collection that may result from this allocation.



Despite these "perplexities", it's worth knowing how much space an expression allocates — how space-intensive it is. It's relatively easy to add this information to the results produced by empg.icn, the expression benchmarking program generator described in the last Analyst. The keyword &storage generates the space in use in each allocated region. It's just a matter of taking differences before and after a benchmarking loop and dividing by the number of iterations to get the average amount of space allocated.

When benchmarking several expressions in the same run, an expression that allocates space may affect a subsequent expression. Even the initialization code to set up a benchmarking run has some effect. Forcing a garbage collection before each benchmarking loop reduces (but may not eliminate) the effects of previous expressions.

If a garbage collection occurs during a benchmarking loop, the time for the loop is likely to be badly distorted and any calculation of the amount of storage the expression allocates is problematical at best. For these reasons, the value of &collections is recorded before the loop and if its value has increased by the end of the loop, a warning is given and the allocation figures are not produced.

All this is reasonably simple, but empg.icn is getting pretty complicated by now; after all, it has to write a program to do all this. We won't try to show this part of empg.icn here, but here's the timing loop it puts out for the first expression given at the beginning of this article:

```
_Prolog()
_Itime := &time
every 1 to _Count do {
   &null & every !&digits
   }
_Epilogue(&time – _Itime)
```

What has to be done before and after the loop now is sufficiently complicated that procedures are used. The procedure to prepare for the loop is:

```
procedure _Prologue()
   _Store := [ ]
   _Coll := [ ]
   collect()
   every put(_Store,&storage)
   every put(_Coll,&collections)
end
```

The lists _Store and _Coll are created to hold the sizes of the storage regions and the number of collections in each region, respectively. Once they are allocated, a garbage collection is forced to free as much storage as possible. Then the current region sizes and garbage collection figures are put into the lists. Since an empty list has space for eight values, adding this information does not cause any allocation.

The processing after the loop is a bit more complicated:

```
procedure _Epilogue(_Time)
  every put(_Store,&storage)
  every put(_Coll,&collections)
  write(fix(real(_Time) / _Count – _Overhead,1,8),
    " ms.")
  if _Coll[1] = _Coll[5] then {
    write("average allocation:",)
      every _I := 1 to 3 do
        write("   ",_Names[_I],
         fix(real(_Store[_I + 3] –
           _Store[_I]),_Count,12))
    }
  else {
  write("garbage collections:")
  write("   total ",right(_Coll[5] – _Coll[1],4))
  every _I := 6 to 8 do write("   ",_Names[_I – 5],
    right(_Coll[_I] – _Coll[_I – 4],4))
    }
  write()
end
```

The procedure fix(), which is from the Icon program library, formats real numbers for printing.

Before anything else is done, the new sizes of the storage regions and numbers of garbage collections are appended to their respective lists. &collections generates four values, the first of which is the total number of garbage collections. Consequently, the first and fifth values of _Coll are the same provided no collection occurred during the loop. In this case, the average amount of space allocated in each region is printed. On the other hand, if a garbage collection occurred in the loop, invalidating the figures for space used, this information is skipped and garbage collection information is supplied instead.

This code is admittedly somewhat unsightly. Most of the problem is due to the need to access the storage region sizes and counts of garbage collections in a different order from their generation. This is a case in which it would be more convenient if &storage and &collections returned lists instead of being generators — except for the fact they would have to allocate space to do so, and hence complicate the storage allocation situation. The code would be cleaner if "before/after" pairs of lists were used instead of putting all the values in two lists. But that would have increased the perturbation of storage, something empg tries to minimize. Ugly or not, the procedures above get the job done.

The output produced by the benchmarking program for the expression

```
every !&digits
```

is

```
0.480 ms.
average allocation:
static        0.000
string       10.000
block         0.000
```

The allocation information should be no surprise. The element-generator operator works on several types directly, but not on csets. Consequently, the cset is converted to a string. That is, a 10-character string is constructed and one-character substrings are generated from it. Note that this conversion is done every time the expression above is evaluated.

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U.S.A.

(602) 621-2018

FAX: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

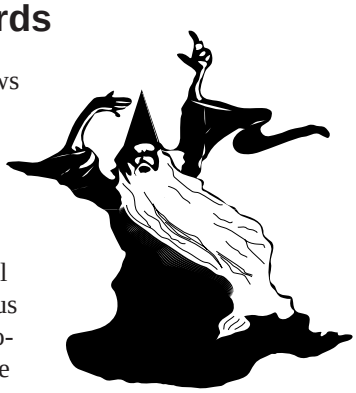
...{uunet,allegra,noao}!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

From the Wizards



A person who knows the innermost secrets of a programming system is called a guru. Similarly, a programming wizard is one who can do amazing if not magical things, things most of us would never think of doing. We'll expose some tricks of the Icon wizards in this occasional feature of the Analyst.

The somewhat arcane, but nonetheless useful programming technique that follows is due to Bill Mitchell, our Wizard of the Issue, as it were.

Have you ever wanted to trace an Icon function (as opposed to a procedure)? ProIcon has a feature for doing this, but the public-domain implementations of Icon do not. Here's a way to "front-end" an Icon function, so that when you call it, you get an equivalent procedure that can be traced.

Let's use abs(n) as an example. There are two things to do: provide a procedure and connect it with the function. Suppose the procedure name is Abs (Icon's differentiation of cases is handy in situations like this). At the beginning of the main procedure, swap the values of abs and Abs:

```
procedure main()
  Abs :=: abs
    .
    .
    .
```

As a result, the value of Abs is the function for computing the absolute value of a number and the value of abs is the procedure that formerly was the value of Abs. Here's the procedure:

```
procedure Abs(n)
  return 2(Abs :=: abs, .abs, Abs :=: abs)(n)
end
```

When the procedure is called, the values of Abs and abs are exchanged again (restoring to their original values at the beginning of program execution), abs is dereferenced to produce the function for computing absolute values, the values of Abs and abs are exchanged again (restoring them to the values they had when the procedure was called), the second value in the mutual evaluation is applied to the argument of the procedure, n, and the result is returned.

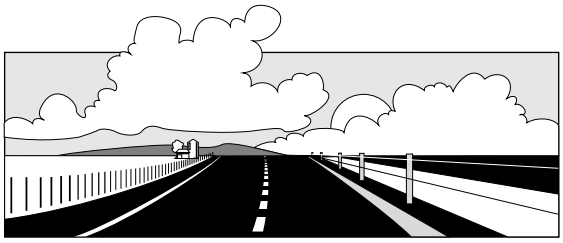
Note that it's necessary to dereference abs to prevent the assignment in the third expression in the mutual evaluation from changing it back before it is used.

Neat, if a bit complicated. It took another wizard, Ken Walker, to see a much simpler formulation:

```
procedure Abs(n)
  return Abs(n)
end
```

That's insight! Since the values of abs and Abs have been exchanged, abs(n) calls the procedure, and Abs(n) in the procedure calls the built-in function for computing absolute values.

This formulation handles functions that may fail. It's simple to change it to handle suspension also.



What's Coming Up

Icon programs often can be run on a wide variety of different platforms. In the next issue of the Analyst, we'll discuss some of the considerations is making Icon programs truly portable.

We'll also give some tips on writing your Icon programs so that they are easy to read.

String scanning is a major and important feature of Icon. Yet many Icon programs aren't really sure how to use string scanning. In the next issue, we'll start a series of articles related to string scanning — a review of the basic concepts, guidelines for writing scanning expressions, how to extend string scanning to pattern matching, and so forth.

As described in the article on expression evaluation, generators provide the alternatives for expressions that seek to achieve a goal. We'll have more to say about generators and their uses in the next issue.

We'll also have a couple more programming tips that may make your programming easier and more efficient.

Please let us know what you think about the Analyst and what you'd like to have us cover in future issues. See the box on page 11 for information about how to contact us.

Back Issues

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

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)
(128.196.128.118 or 192.12.69.1)