The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1991
Number 4

In this issue …

Programs that Write Programs … 1
Writing Scanning Expressions … 2
Large Integers … 5
Memory Utilization … 7
From the Wizards … 10
Programming Tips … 11
Thanks and Credits … 11
Feedback … 12
What's Coming Up … 12

Programs that Write Programs

There's something fascinating about a program that produces another program, even if the result is simple. It seems like an act of creation, even if it isn't. Programs that write programs in the creative sense are rare; such cases usually are so complicated that the act of creation, if any, is lost in complexity. There's an argument that creativity is nothing more than the right kind of complexity, but our concerns are more mundane.

Even in the absence of a creative act, there are metalinguistic issues with programs that write programs, and these produce syntactic problems. Such problems arise whenever it's necessary to talk about language; you have to distinguish the language you're using for description from the language you're describing. If both languages are the same, it can get confusing and complicated.

Icon has only one mechanism for distinguishing strings from program text: surrounding quotation marks. The use of "x" to distinguish the string x from the identifier x is so natural and easy that you probably never make a mistake in writing the two, except maybe for failing to provide the closing quotation mark for a long string.

When a program writes a program, however, things get sticky. It's easy enough to write out a program line such as

```
write("write(&time)")
```

— you're writing the string

```
write(&time)
```

However, if the argument of the (inner) write() is a literal string such as "end", you can't just use

```
write("   write("end")")
```

The Icon compiler thinks " write(" is a string, and when it then encounters end, it reports a syntax error, since it's syntactically incorrect to have a literal immediately followed by an identifier.

Icon provides several ways around such problems. The most straightforward way is to use its escape mechanism, in which the character immediately following a backslash is interpreted in a special way. If a quotation mark immediately follows a backslash, the two characters are interpreted as a literal quotation mark instead of the quotation mark serving as the end of the literal. Consequently, the expression above can be written as

```
write("   write(\"end\")")
```

and the string written out is

```
write("end")
```

The trouble with this technique is that quotes and escaped quotes get confusing. While it's easy for a computer program, such as Icon's compiler, to keep them straight, the human mind isn't built that way. There are a few things you can do to make it easier to handle the "meta-quote" problem. One approach is to assign a string consisting of a double quote to an identifier and then use the identifier wherever a quote is needed:

```
quote := "\""
write("   write(",quote,"end",quote,")")
```

There are two problems with this approach. One is that a single string has been replaced by five strings. There's no significant loss of efficiency in this case, since write() just writes its string-valued arguments, one after another. If, however, the situation is changed slightly so that the string has to be retained in the program, actual concatenation is needed:

```
line := "   write(" || quote || "end" || quote || ")"
```

Such constructions are awkward to write and hard to read (although they probably are not as bad as escaped quotes).

It's really a matter of taste as to how you chose to represent literal quotes. If you're good at handling nested

syntax (and don't expect persons who aren't good at it to read your programs), the escaped quote sequence is the shortest and most direct.

There's another device that's helpful on occasion: the function image(). If its argument is a string, its value is that string with surrounding quotes and any internal quotes given with escape sequences. Thus, the example above can be written as

```
write("   write(",image("end"),")")
```

This device takes a little getting used to, but once you see what's going on, you'll realize there are situations (not the one above) where it's very useful indeed.

There's a short program in the Icon program library, iwriter.icn that uses just this technique:

```
procedure main()
  while write("write(",image(read()),")")
end
```

To see why you might want iwriter.icn, consider what's involved in empg.icn ("Benchmarking Icon Expressions", The Icon Analyst, Issues 1 and 2). This program has to write fairly complicated Icon code. Think about knowing the kind of code you want to write and working backward to a program that writes it.

Now consider a little puzzle — writing a self-replicating Icon program that, when executed, writes out a copy of itself. Better yet, write the shortest possible such program.

The shortest known self-replicating Icon program to date was written by Ken Walker and appears in the box below. Can you do better? Or prove the one shown below is the shortest possible?

```
procedure main();x:="procedure main();x:= \nx[21]:=image(x);write(x);end"
x[21]:=image(x);write(x);end
```

Shortest Known Self-Reproducing Icon Program

Writing Scanning Expressions

This is the second in a series of articles on using string scanning in Icon. This article deals with some pragmatic matters.

When to Use String Scanning

Since Icon has low-level string-analysis operations in addition to string scanning, you may have trouble deciding when to use low-level operations and when to use string scanning.

Consider an example. Suppose a program reads lines from a file. Most of the lines are data, perhaps to be formatted, but if a line begins with an exclamation point, the program calls system() with the balance of the line as the argument. A low-level approach to the handling of these "special" lines is:

```
while line := read() do {
  if line[1] == "!" then system(line[2:0])
  else …
  }
```

A string-scanning approach to the same problem is:

```
while line := read() do {
  line ? {
    if ="!" then system(tab(0))
    else …
    }
  }
```

Which approach is better? While there's no clear-cut answer to this question, it's generally better to use string scanning, even in simple cases like this one. String scanning is almost certainly better for even moderately complicated string analysis, and simple code tends to get more complex as a program evolves. Using string scanning for all analysis produces a consistent style.

Most programmers who use low-level string analysis say they do so because it's faster, although some will admit that they are not comfortable with string scanning.

The assumption that low-level analysis is faster than string scanning is an example of how easy it is to make the wrong conclusion about the performance of Icon. The fact is that string scanning is almost always faster than low-level analysis. Even in the simplest expressions, string scanning typically is 50% faster than the equivalent low-level analysis. Why? In the first place, there's very little overhead in getting into string scanning. And, once you're there, the operations are cast at a higher level with less overhead and fewer elementary computations are needed. This is one case where "low-level" doesn't translate into "efficient".

It's natural for low-level string-analysis operations to be easier to use when you are first learning Icon, since these operations are similar to operations in other programming languages that you may already know. Such "inertia" is a major factor in many programmers' choice of style and approach.

Once you get used to string scanning, and especially if you use it consistently in favor of lower-level operations, you'll find your investment will pay off handsomely.

Of course, like any such advice, the recommendation to use string scanning needs to be applied intelligently. It makes sense if the low-level operations and string scanning are performing the analysis in the same way. But it's not a good idea, for example, to try to use string scanning to see if a string is palindromic (that is, if it reads the same forward and backward); the function reverse() is a much better choice.

Perhaps a better way to phrase the recommendation is to use string scanning if the problem can be naturally cast that way. And, of course, low-level operations have their place. You'd not want to use string scanning in place of expressions like

```
return s[1:–1]
```

and

```
s[i] := ""
```

Formulating Scanning Expressions

Assuming we've talked you into at least trying string scanning, you may wonder where to start. One of the problems with string scanning is that analysis expressions often are fairly complex. This is to be expected, since one of the conceptual bases for string scanning assumes that string analysis consists of a succession of operations. String analysis expressions often are "little programs", but since they also are just Icon expressions, there's a tendency to write the subject, a question mark, and then just plunge into the analysis expression. Before long, however, the analysis expression is likely to become awkward and poorly structured.

It's usually worth spending a little time thinking about an analysis expression before starting to code. One device that's helpful in enforcing this discipline is to encapsulate the analysis expression in a procedure. For example, a scanning expression to write out all the words in a string can be written as

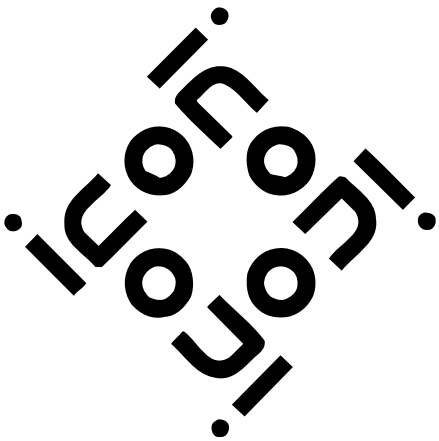
```
sentence ? while tab(upto(&letters)) do
  write(tab(many(&letters)))
```

But it might be better to write it as

```
sentence ? writewords()
```

with the procedure

```
procedure writewords()
  while tab(upto(&letters)) do
    write(tab(many(&letters)))
end
```



Note that the call of writewords() does not affect the scanning environment; the subject and position are the same inside the procedure call as outside.

The encapsulation of analysis expressions in procedures is helpful in organizing programs. Encapsulation also is helpful if the procedures need to be modified; think about a more sophisticated definition of words in the example above. If an analysis expression is encapsulated in a procedure, it also can be used in different scanning expressions without having to duplicate it or modify multiple copies.

Procedural encapsulation, however, requires extra work and discipline, and most programmers are not willing to use it consistently as a programming technique. A simpler device, one that at least encourages you to stop and think, is to always open a scanning expression with a brace, even if you think it's not going to be necessary. The use of braces also makes the scope of the analysis expression easier to see, as in

```
sentence ? {
  while tab(upto(&letters)) do
    write(tab(many(&letters)))
  }
```

In addition to improving program readability, it prevents mistakes like

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

where pos(–1) is intended to check that the period is at the end of the subject. As noted in Issue 2 of the Analyst, this expression groups as

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

so the pos(–1) applies to the outer scanning environment. It usually doesn't take many such experiences to convince an Icon programmer to always use braces:

```
sentence ? {
  tab(upto('.')) & pos(–1)
  }
```

Style Considerations

Poorly structured analysis expressions are a common cause of problems in using string scanning, especially for novice Icon programmers. Here are a few suggestions for avoiding such difficulties:

- Avoid low-level operations in analysis expressions; stay on one level of thinking. An example of bad style is:

```
sentence ? {
  while tab(many(&letters)) do {
    word := tab(many(&letters))
    if word[1] == "X" then write(word)
    }
```

A better formulation is:

```
sentence ? {
  while tab(many(&letters)) do {
    word := tab(many(&letters))
    if word ? ="X" then write(word)
    }
```

Note that there's nothing wrong with nesting scanning expressions; the whole reason for maintaining scanning environments, as described in Issue 3 of the Analyst, is to make sure expressions such as this work the way you'd expect.

• Don't refer to &subject and &pos in analysis expressions unless it's really necessary. For example, use tab(i), not &pos := i, to set the position. The whole idea behind string scanning is to avoid explicit references to the subject and position. Incidentally, tab(i) is faster than &pos := i. Another example of bad style in string scanning is:

```
line ? {
   if ="!" then write(&subject[2:0])
   }
```

It's better to use

```
line ? {
   if ="!" then write(tab(0))
   }
```

• If the subject expression is a variable, don't use that variable inside the analysis expression. It's not unusual to see poorly conceived scanning expressions such as this:

```
line ? {
  if ="!" then write(line[2:0])
  }
```

Again, tab(0) is better.

• As mentioned above, always enclose analysis expressions in braces.

• Write complicated analysis expressions on several lines as necessary to make them readable.

• Encapsulate complicated analysis expressions in procedures.

Scanning Paradigms

There are no syntactic restrictions whatsoever on what kinds of computations can be put in an analysis expression. This can be very useful; you can write data from inside a scanning expression, perform arithmetic, and so forth. On the other hand, this freedom tends to get Icon programmers in trouble. A little discipline goes a long way here.

One road to discipline in writing scanning expressions is to think of them in terms of models that have some structure. There are two useful paradigms for structuring analysis expressions: iterative scanning and pattern matching.

In iterative scanning, the position moves along the subject, generally from left to right, and portions of the subject are processed as they are encountered. The scanning expressions given above to write the words in a sentence are typical of this paradigm.

In pattern matching, expressions are used in conjunction to assure the mutual success with backtracking to find alternatives. A problem that is typical of the pattern matching paradigm is identifying outer parentheses in a parenthesis-balanced expression:

```
expression ? {
  ="(" & tab(bal()) & =")" & pos(0)
  }
```

The analysis expression succeeds if the subject consists of a left parenthesis followed by a parenthesis-balanced string, followed by a right parenthesis that is at the end. It's worth noting that the analysis expression may involve data backtracking. Suppose, for example, the value of expression is "((a+b)+(c+d))". The expression tab(bal()) first matches "(a+b)", but since this is not followed by a right parenthesis, tab(bal)) is resumed to produce "(a+b)+" and then "(a+b)+(c+d)".

As a point of style, conjunctions like this are easier to read if the components are written on separate lines:

```
expression ? {
  ="(" &
  tab(bal()) &
  =")" &
  pos(0)
  }
```

Contrast this scanning expression with the following one:

```
expression ? {
  if ="(" then {
    if tab(bal()) then {
      if =")" then pos(0)
      }
    }
  }
```

This expression may look like it will work, but it won't, since tab(bal()) is used in a bounded expression where it can't be resumed to produce another value (see the example mentioned earlier). We'll have more to say about this later in an article on pattern matching.

In summary, iterative scanning stresses the process of scanning and is appropriate where portions of the subject can be processed in sequence and then "discarded". Pattern

matching stresses the relationships among components of the subject and is appropriate where the desired analysis needs to be phrased in terms of the overall structure of the subject.

Iterative scanning is, in some sense, lower level than pattern matching. It tends to be imperative — "do this then that". Pattern matching allows a higher level of abstraction and tends to be declarative — "this kind of string followed by that kind of string". Pattern matching is a very powerful tool and is particularly useful in the analysis of complex string structures.

Iterative string scanning and pattern matching are in some sense extremes. Many analysis expressions combine aspects of both. The two approaches are, nonetheless, fundamentally different and it's worth giving some thought to the nature of a complicated string-analysis problem before plunging in. These paradigms can provide guidance on the best approach to take.

Large Integers

Version 8 introduced large integers to Icon — integers not restricted in magnitude by the limitations of ordinary computer architecture. Prior to that, Icon integers were limited by two's-complement 32-bit arithmetic and hence to the range -2^{31} to $2^{31} - 1$ (–2,147,483,648 to 2,147,483,647). (A few implementations of Icon use 64-bit arithmetic with a correspondingly larger range.)

Large integers seem natural to Icon. After all, Icon allows arbitrarily long strings, limited only by the amount of memory available. Why should integers be limited to a fixed range?

There are reasons for limiting integers to what a computer can handle directly. One is speed. Another is the difficulty of doing what's called "arbitrary-precision arithmetic" in software. We prefer the term "large-integer arithmetic" and use it here.

Implementing large-integer arithmetic really is difficult, especially if it's done well, completely, and efficiently. There are some tough policy decisions and lots of little things to worry about. These problems are not new, however, and several programming languages, notably dialects of Lisp, have supported large-integer arithmetic for a long time.

Icon presents some special problems in implementation because of its automatic type conversion and dynamic memory management. Fortunately, the code for large-integer arithmetic in Icon was donated by a person who was familiar with the facility in a Lisp system. It didn't come totally free — a lot of work was needed to retrofit it to Version 8 of Icon and to make it work with computers with 16-bit processors. Even now there are a few rough edges, but it's basically a high-quality and robust implementation.

But what are large integers good for? 32-bit integers are enough for counting and representing the sizes of most objects. There are, of course, a host of number-theoretic problems that deal with very large integers. As of this writing, for example, the largest known prime is $391{,}581 \times 2^{216{,}193} - 1$ and has 65,087 digits in its decimal representation. That's a big number, but not especially large by number-theoretic standards. Clearly, if you are interested in problems in number theory, you can't live with 32-bit integer arithmetic.

It's worth noting that if you don't use large integers, their presence in Icon will not bother you or affect the performance of your Icon programs. Integers that can be handled directly by computer hardware are represented and manipulated as before. Large integers only come into existence if the result of an arithmetic computation won't fit in the range of "native" integers.

Large-integer arithmetic seems to be an emotional issue. Some programmers couldn't care less about it, while others light up and run off to compute 1,000! just for the fun of it. We'll suppose for the moment that you have enough interest in large integers to read on.

Speed and Space

Large-integer arithmetic is not without its problems. They are the usual ones in computing: speed and space. You can't expect the division of two large integers to be fast. If you don't already know the time complexity of various arithmetic operations, you may want to review the reference at the end of this article before undertaking a significant project involving large integers. A number with a lot of decimal digits also requires a lot of memory. There also are a few kinky aspects to the implementation that you need to know about if you're going to use large integers.

In most cases, the time required to perform an operation that produces a really large integer pales in significance compared to the time it takes to convert the large integer to a string so that you can see it or print it out. For example, computing the largest prime mentioned earlier takes only about two minutes on a Sun Sparcstation, but it takes more than two *hours* to convert the result to a string. The problem is not one of poor implementation; it's fundamental. The best known algorithm for converting the internal representation of an integer to a string (which is an instance of radix conversion) is quadratic (n^2) in the size of the integer. For example, 10^{1000} takes on the order of 100 times longer to convert to a string than 10^{100}. Not only does it take a long time to convert a really large integer to a string; there's a psychological problem — you'll probably wonder if the conversion really is taking all that long or if your program (or Icon) is in a loop.

The quadratic nature of integer-to-string conversion poses implementation problems too. If your program writes large integers, that's presumably what you intended. If, however, you turn on procedure tracing to see what's going on and

an argument or returned value is a large integer, are you prepared to wait minutes or even hours for every trace message to perform the conversion? Presumably not. For this reason, the string image of an integer with more than about 25 digits shows only the approximate number of digits in the corresponding string, not the actual string itself. For example, image(10 ^ 100) produces "integer(~99)". The number of digits is only approximate; figuring out the exact number of digits amounts to doing the whole conversion.

Space problems come in two forms. One problem is the amount of memory needed to store large integers. Large integers are stored as blocks of "digits", where a digit usually is a 16-bit (native) integer. There is some overhead, but you can count on less than two bytes per decimal digit for really large integers. The string space for the decimal representation of a large integer cannot be ignored; it's just one byte per digit. These figures should tell you one thing — you're not going to be able to compute or print the largest known prime on an MS-DOS implementation of Icon, where the string and block regions are limited to 65,000 bytes.

There's a more general pragmatic consideration. It takes a lot of code to support large-integer arithmetic. It typically increases the size of Icon's run-time system by about 15%. Icon's run-time system already is large, and memory problems are severe for many personal-computer users. This increase may render Icon useless even to persons who have no need for large integers. Consequently, large-integer arithmetic is not supported for all personal-computer implementations of Icon. (For MS-DOS, two run-time systems are included in the distribution, one with large-integer arithmetic and one without it.)

Programming Considerations

Internally, there are two representations of integers: *machine* integers that are handled in the mode that is native for the computer and *large* integers that are stored as blocks of digits as mentioned earlier. The difference in representation is largely invisible; both have type integer, and there's no source-language test to find out what the actual representation is. A large integer only comes into existence if machine-integer arithmetic would overflow. (Icon did overflow checking before large integers were added, so this is nothing new.) Conversely, arithmetic on large integers that produces a result in the range of machine integers produces a machine integer.

Despite the "transparency" of large integers, there a few things you should know about the implementation of large-integer arithmetic if you intend to use it.

While large integers are supported in most operations, there are a few holes. Large-integer arithmetic is not supported for i to j by k or in seq(). In addition, large integers cannot be assigned to keywords, such as &trace, that require integer values. It's even possible there is some obscure numerical nook that we overlooked that should support large integers but doesn't.

There's one implementation "hack" you should know about if you need large-integer literals in your programs. Such literals are not converted to blocks of digits by Icon's compiler or linker. Instead, they are passed through to the run-time system as strings but flagged to be converted automatically to large integers when they are evaluated. The conversion process is transparent, but it takes time and allocates space. Avoid using large-integer literals in loops where conversion may take place repeatedly.

There's also a conceptual problem with treating a large integer as a bit string and shifting the bits using ishift(i,j). In two's complement arithmetic, the most-significant bit is one for negative numbers. Consequently, if you shift a negative integer right one bit and fill the vacated position by zero, the integer becomes positive. Conversely, if you shift a positive integer to the left far enough, it becomes negative. But what about large integers, which don't have this machine representation? There seems to be no consistent way to treat bit shifting of large integers that satisfies everyone's expectations. In Version 8 of Icon, shifting the bits of a negative integer right results in a positive integer, but shifting the bits left does not result in a negative integer — once an integer is positive, it stays that way under the operation of bit shifting.

Finally, there one known bug in large-integer arithmetic: Overflow during exponentiation of machine integers may not be detected and may result in erroneous integer values.

The Bottom Line

Don't let these problems scare you off. We've listed them so you'll know they are there, but they only affect operations that rarely occur in practice.

If you're interested in trying large integers but don't happen to have an unsolved number-theoretic problem handy, try this one: Take any positive integer, reverse its digits (that is, reverse the string representation of the integer), and add the result to the integer. If the result is palindromic (that is, if it reads the same from left to right as it does from right to left), stop. Otherwise continue the process with the new integer.

You'll find this process terminates quickly for most integers. For example, 169 → 1130 →1441. But for some integers, such as 196, the process does not appear to terminate. In fact, whether or not this process terminates for all integers is an unsolved problem.

Reference

Knuth, Donald E. *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, Addison-Wesley, Reading, MA., 1969, pp. 229-290.

Memory Utilization

Strings and structures require a lot of memory. Icon is a big program. Many computers have only comparatively small amounts of memory. All this adds up to trouble.

If you are using Icon on a workstation, super-mini, or mainframe where you're likely to have ready access to several megabytes of memory, what follows may be of little interest to you. If you're running Icon on a personal computer with a small amount of memory, you're bound to run into trouble sooner or later. If the amount of memory you have really is inadequate for Icon, you may be having trouble running at all.

There is a more frustrating situation: You may invest a lot of time and effort in developing an application in Icon, only to discover that when you try to run it on *real* data, there isn't enough memory; your program doesn't scale up.

This article contains information on how Icon uses memory and what you can do to minimize its memory requirements. If you're worried that a program may not scale up, there's information here to enable you to get a handle on what to expect.

Planning Ahead

Icon often provides several different ways of doing the same thing, sometimes with different data structures. It's a plain and simple fact that your choice of data structures may make a lot of difference in the amount of memory your program needs — and hence, in some situations, whether it will work or not.

Icon programmers often are advised (usually by persons with lots of memory on their computers) not to worry about what data structure to use, but to do what seems easiest. This is good advice in many situations, but taking it at face value can lead to frustration and unnecessary work.

If you're working in an environment with a limited amount of memory (less that two megabytes, say) and you need to keep a lot of data in memory while your program runs, you are better advised to do some planning.

Unfortunately, determining how much memory a program needs often is complicated, and accurate calculations may be impractical. Nonetheless, knowing how memory is used and how big Icon's data objects are can help you select reasonable strategies for memory-intensive programs.

Run-Time Memory Allocation

Most concerns about the utilization of memory relate to what goes on at run-time, when a program is executed.

Most of run-time memory is needed for the following purposes:

- the operating system itself
- memory-resident programs like device drivers
- the Icon executor, iconx
- the icode file for our program
- Icon's allocated data regions
- I/O buffers and other space needed by the operating system

On personal computers, the operating system and memory-resident programs may take up a significant portion of the available memory. This subject is beyond the scope of this article, but the first thing you should do if you're programming on a computer with a small amount of memory is to find out how much space is available for executing programs. There are various utilities for getting this information. You may be surprised how little memory really is available and you may discover you have memory-resident programs that you didn't know you had and that you don't need.

The Icon executor contains the code that's needed to run an Icon program. You can get an idea of how much memory it requires by looking at its file size. The executor is called iconx, possibly with some suffix like .exe. The amount of memory needed is not necessarily the same as the file size (the file may be compressed or may carry excess baggage), but it's a good first approximation.

An icode file results from compiling and linking your Icon source program. The icode file is loaded into memory when the Icon executor starts up. The size of an icode file depends on a lot of things, but it's largely a function of the size of your Icon source program and typically is 3 to 4 times the size of the corresponding source code with white space (notably comments) removed. To find out how much memory is needed for the icode file, just look at its file size (icode file names are the same as source-code file names, but with the .icn suffix removed or replaced by .icx). There's an exception: on UNIX systems, the icode file has a bootstrap header that loads iconx automatically. The size of this header, which varies from system to system, should be deducted from the icode file size in determining how much memory the icode file will occupy. We'll have more to say about what affects the size of an icode file in the next section.

Icon's allocated data regions take a lot of memory — they provide the space for data objects that are created during program execution. There are several regions. The major ones are the string region, the block region, and the evaluation stack. As the name suggests, strings created during program execution are stored in the string region. All other objects created during program execution (notably structures) are stored in the block region. The evaluation stack provides space for temporary results produced during expression evaluation.

The string and block regions normally occupy 65K bytes each and the evaluation stack normally is 40K bytes. You can make the region sizes larger or smaller by using environment variables, although the maximum region size may be limited by computer architecture.

Some implementations of Icon allow region sizes to grow if more space is needed as a program executes. Most

personal-computer implementations of Icon, where memory problems tend to be the most severe, do not allow region sizes to grow. We'll assume here that Icon's allocated regions cannot grow.

In addition to Icon's allocated regions, which are established when iconx starts up, space may be needed during program execution for I/O buffers and other operating system uses (such as the execution of Icon's system() function). This space is taken from whatever is left over after iconx, the icode file, and Icon's allocated regions are in memory. Except for implementations that allow Icon's allocated regions to grow, space for co-expressions also is taken from whatever is left over.

Just to get a handle on how all this adds up, consider a typical MS-DOS system. MS-DOS itself and memory-resident programs probably take at least 100K bytes (the figure may well be more). The Icon executor takes about 200K. The allocated data regions take about 170K. That adds up to 470K out of the 640K available under MS-DOS — leaving 170K for everything else. A good-sized icode file may be 30K (there's a 65K limit under MS-DOS). That leaves 140K, which is enough for some co-expressions, I/O buffers, and so forth. Whether system() will work in what's left over is problematical. If MS-DOS itself and memory-resident programs take more than 100K, the amount of free memory is correspondingly squeezed down. It's no wonder that persons running MS-DOS have trouble with large Icon programs.

One thing about this MS-DOS scenario that is fundamentally important is the 65K limit on region sizes. This limit is a consequence of the segmented i86 architecture and the limitations of C compilers. 65K regions are not big enough for really memory-intensive programs. If you have to operate in this environment, you'll have to give a lot more consideration to what you can do and how you're going to do it than if you are running on a workstation with several megabytes of memory (real or virtual).

The balance of this article is devoted to information that may be helpful to you if you have a "tight" memory environment.

The figures given are for typical implementations of Icon and apply to computers with 16- and 32-bit integers. For computers with 64-bit integers, most values are twice as large except for strings, which are the same size on all implementations.

Icode Files

There is not a great deal you can do about the size of an icode file, short of keeping your program small or dividing its functionality among several programs. If, however, the size of an icode file is a problem (as, for example, in approaching the 65K barrier in MS-DOS), it may be handy to know a few things about what's in an icode file and how much space it takes.

Since an icode file is a binary version of a source program, it must necessarily contain all the information necessary to run the program. In addition to compiled code, it contains all constants: literal strings, csets, and real numbers. (Integer literals are embedded in the compiled code.) An icode file also contains information about declarations: procedures and records.

Every string that appears in a program (except for comments) is stored in a string region of its icode file. This includes the names of identifiers as well as string literals. These strings are pooled for the entire program; no matter how many times the same string appears or how it is used (for example as an identifier and as a literal), it appears only once in the icode file's string region. Strings are packed and null-terminated, so the amount of space needed for strings in an icode file is the total number of characters plus one byte for each different string.

Cset and real literals are pooled, but only on a per-procedure basis. The space required for such literals on a per-procedure basis is:

cset literals	40 bytes
real literals	12 bytes

Space also is required for identifiers: global identifiers (including procedure and record names), static identifiers, and local identifiers (which include arguments). The space required in an icode file for each distinct identifier is :

global	16 bytes
static	16 bytes
local	8 bytes

Each procedure requires 72 bytes in addition to the code it contains (the space for arguments and local identifiers is included in the figures above).

Each record declaration requires 52 bytes for its record-construction function. In addition, there is an $n \times m$ matrix, where n is the number of record declarations and m is the number of distinct field names.

While it's probably not worthwhile to worry too much about these details, if icode file size is a problem, you can reduce it by minimizing the number of global variables and possibly the number of procedures, although the use of procedures for organizing a program probably is more important than the amount of space they take in a icode file.

Since the size of the matrix for record field names depends on the product of the number of different records and distinct field names, if your program has many different record declarations and a lot of different field names, you should look to this as a source of excessive icode file size.

Storage Allocation

For most programs, the main consideration in memory utilization is the amount of space needed for objects that are allocated during program execution. In particular, if these

objects are retained in memory over the course of program execution, instead of being transient and hence collectable, your program may suffer degraded performance (due to frequent garbage collections that do not free much space). It even may not have enough space to run to conclusion.

A complete description of the amount of space required for objects created during program execution is very complicated and is beyond the scope of this article. See References 1 and 2 for more information. What follows are the highlights; they should be enough to help you chose the best data structures to use for your programs and provide the necessary information for deciding what to do if you have memory utilization problems.

Strings created during program execution are stored in the allocated string region. They are packed but not null-terminated. So count one byte of storage for every character in every string created by your program. It's worth noting that substrings do not require any additional storage. And, of course, space for strings that are no longer in use is reclaimed by garbage collection.

The space for csets and real numbers created during program execution is the same as the amount of space in an icode file: 40 bytes and 12 bytes, respectively.

Structures created during program execution are the usual cause of memory utilization problems. To understand structures, it's necessary to understand how Icon values are represented.

Since any variable in Icon can take on any kind of value, there must be enough space associated with every variable to hold any kind of value. Some values can be arbitrarily large, so they cannot be stored in a fixed-size space. Instead, they are referenced indirectly by pointers. Every Icon value is represented by a *descriptor*, which either contains the value itself or a pointer to it. Descriptors also contain other information, such as the type of the value. Every descriptor requires 8 bytes of storage. For example, every identifier requires 8 bytes of storage for its value. The space required for identifiers is accounted for by the figures given for icode files in the preceding section.

Structures also contain values. These come into being when the structure is created or when elements are added to it. For example, list(10) creates a list with 10 elements and consequently contains 10 descriptors (which require 80 bytes of storage).

The space for descriptors contained in a structures is one consideration in understanding the memory requirements of structures. Icon's structures consist of more than just the descriptors, however ; the sophisticated access mechanisms provided for structures require additional space. In addition, automatic storage management and the information needed for garbage collection add additional space overhead for structures. The details are moderately complicated for lists, sets, and tables, since they can grow and shrink in size and hence their size at any particular time may depend on the history of access to them. The figures that follow are first approximations to the memory requirements for structures. See References 1 and 2 for additional information.

A list that is fixed in size has 48 bytes of space overhead in addition to 8 bytes per element. For example, list(1) allocates 56 bytes, while list(10) allocates 128 bytes.

Empty lists are exceptions. It is presumed that an empty list is created with the intention of adding values to it by queue or stack access, and it is created with space for 8 values. That is, list(0) and list(8) allocate the same amount of space, 112 bytes. The first 8 elements added to an empty list come free. If more are added, additional space is allocated. The computation of the exact amount is somewhat complicated, since it depends on how big the list is — the bigger the list is, the more space is provided for additional elements, so as to reduce the overall storage overhead. See Reference 2 for details.

Sets and tables are even more complicated, since space is used to provide efficient look-up. Both sets and tables have header blocks and space used to reference elements. An empty set requires 96 bytes, while an empty table requires 104 bytes (the difference is a descriptor for the default value for the table).

Set and table elements require more space than just that needed for their values. Both have overhead related to efficient look-up, and every table element has a key descriptor as well as a value descriptor. Every set element requires 20 bytes and every table element requires 28 bytes. As tables and sets get large, there is increased space overhead for allowing efficient access. See Reference 2 for details.

Records are simple. A record with n fields requires $16 + (8 \times n)$ bytes.

To summarize the approximate storage requirements, in bytes, for structures with n elements:

lists:	$48 + (8 \times n) + \Theta$
sets:	$96 + (20 \times n) + \Theta$
tables:	$104 + (28 \times n) + \Theta$
records:	$16 + (8 \times n)$

where Θ indicates some additional space overhead that depends in a complicated way on the size of the structure and its access history. For large structures, however, the amount of space that depends on n dominates this overhead as well as the constant term.

Some general ideas about the space requirements of alternative structures can be seen at a glance. For example, for a large number of elements, a set requires about 2.5 times as much space as a list. In many cases, it's possible to use a list in place of a set. Of course, if a set is the natural way to deal with your data, you get a lot in return for the extra space required: fast tests for membership and a fast way to delete an element, not to mention the operations of union, intersection, and difference. Most important, you can think about a collection of values in set terms.

Another thing to think about is that strings often provide a very compact representation of data. You often can encode a lot of information in a string with little space overhead — perhaps just a few punctuation characters here and there. For example, a tree can be encoded in a string by using parentheses to delimit subtrees and commas to separate nodes at the same level. Accessing the information encoded in this way is, however, likely to be more difficult and unnatural than accessing a tree composed of records. We'll have more to say about this in an upcoming article on space/time trade-offs.

Measuring Allocation

If you want more information about the space requirements for Icon values, you can read the suggested references or you can get your own empirical results. The program empg.icn in the Icon program library is designed to do this. See "Benchmarking Icon Expressions" in Issue 2 of the Analyst. To avoid garbage collection from interfering with the results, use a small value for the number of iterations — 1 will do for finding out how much space a specific structure takes.

For more complicated situations, use the Icon keywords that are designed for providing storage information. The keyword &collections generates garbage collection information; first the total number of collections to date, and then the number of collections triggered by allocation in the static, string, and block regions. The keyword &regions generates the sizes of the static, string, and block regions, respectively, while &storage generates the amount of space currently in use in the static, block, and string regions. All values are given in bytes. The static region does not exist in many implementations of Icon and the figures given for it generally are not meaningful, so they should be ignored.

References

1. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986.

2. *Supplementary Information for the Implementation of Version 8 of Icon*, Ralph E. Griswold, Icon Project Document 112, Department of Computer Science, The University of Arizona, 1990.

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)

From the Wizards

Wizards do wonderful and magical things. They also, on occasion, do demented things.

The other day we were thinking about radix conversion. Converting an integer i in base j to base 10 is easy — Icon's radix literals provide the notation and conversion of a string representing a radix literal to an integer does the trick:

```
integer(i || "r" || j)
```

What about converting an integer i in base j to base k, where k is not necessarily 10? The usual technique is to first convert the integer to base 10 and then use a conventional chug-and-plug loop, concatenating remainders on successive divisions by 10. See radcon.icn in the Icon program library for this method.

But we thought there might be a more "Icon-ish" way to do general radix conversion and posed our problem to a wizard friend. No doubt influenced by his experiences at a recent Constraint Convention, he said

> That's easy. It's just a matter of solving
>
> (j || "r" || i) = (k || "r" || *x*)
>
> That is, finding the value of *x* for which this comparison succeeds. All we need is a source of values in base k and we can sit back and wait for the right one to show up. We'll use a recursive generator — after all, this is Icon. Let's see … Icon represents digits in radix literals by 0, 1, … 9, a, b, … z. This ought to do:

```
procedure k_integer(k)
   static chars

   initial chars := (&digits || &lcase)[2:0]

   suspend 0 | nstar(chars[1:k])
end

procedure nstar(s)
   suspend !s | (nstar(s) || (0 | !s))
end
```

> Sure enough, here's a general radix conversion procedure:

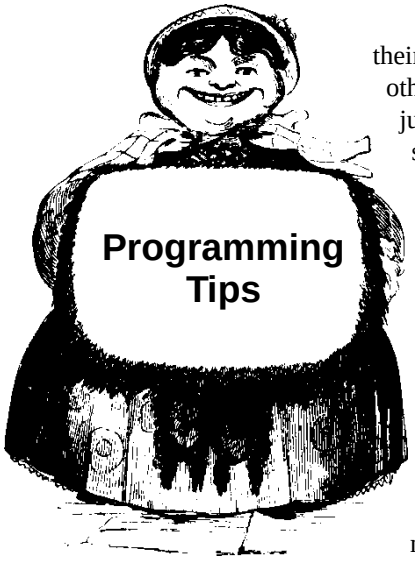
```
procedure radcon(i,j,k)
   local x

   (j || "r" || i) = (k || "r" || (x := k_integer(k)))
   return x
end
```

At this point, our wizard went hopping off into the Arizona sunset, cackling all the way. Wizards tend to do that around here.

His procedure works, and it's even reasonably efficient for converting small values (the bases don't matter much). But it does it by starting at zero and counting up in base k with strings. It takes *forever* for large integers.

Incidentally, we think the wizard's procedures for generating integers in base k can be improved, but after this experience, we haven't the stomach for it.



Generators produce their results one after another. Sometimes this is just what you want, but sometimes it isn't. Suppose you want just the n^{th} result in a sequence produced by a generator. What's the best way to do it?

One way is to put all the results from the generator in a list and then use the n^{th} element of the list, as in

```
L := [ ]
every put(L,expr)
x := L[n]
```

where *expr* is the generator. This method may require a lot of space if *expr* produces many results — and most of that space is wasted. Of course, if *expr* produces an infinite number of results, this is no method at all.

Another method is to use a co-expression, as in

```
C := create expr
every 1 to n – 1 do @C
x := @C
```

This solution works even if *expr* potentially has an infinite number of results, but a co-expression takes a lot of space and it's not really necessary.

A better method to is to use limitation, as in

```
every (x := expr) \ n
```

Here, each result produced by *expr* is assigned to x, the generator is stopped after n results, and the value of x at the point is the desired one.

Note that the three methods produce different results if *expr* does not produce n results. In the list and co-expression methods, the assignment to x fails if there aren't n results, leaving its value unchanged. In the limitation method, the value of x is the last result generated, if any. In the list and co-expression methods, it's easy to add a test for failure. In the limitation method, it's more awkward; a counter incremented in a do clause is a possibility. Of course, you may know that *expr* produces at least n values.

Here's an example of where you might want to use the limitation method. Suppose you want to know how much storage is in use in the allocated block region. This is the third result generated by &storage, so

```
every blocks := &storage \ 3
```

does the trick.

Similar situations sometimes come up in string scanning. Suppose a string contains a list of words, each followed by a comma, as in

```
wordlist := "This,is,a,list,of,words,"
```

One way to get the n^{th} word is:

```
wordlist ? {
  i := 0
  every i := upto(',') \ (n – 1 )
  tab(i + 1)
  word := tab(upto(','))
  }
```

We'll leave it as an exercise to handle the situation in which commas are used as separators and there is no comma after the last word.

See also the related programming tip in Issue 2 of the Analyst.

Thanks and Credits

We rely on two expert readers, Gregg Townsend and Ken Walker, for checking the contents of the Analyst. Their help is invaluable; not only do they find typographical errors, but they also catch potential blunders and often suggest significant improvements.

Any remaining mistakes are, of course, our responsibility.

We use clip art from various sources to add a little visual interest to the Analyst. The graphics used in the "Programming Tips" are called mortised cuts and were used in early nineteenth-century advertising art. The originals were done as wood engravings and carried advertising messages, which we've replaced.

The mortised cuts we use are scanned from *Handbook of Early Advertising Art*, Clarence P. Hornung, Dover Publications, 1956. This book is just one of many in Dover's copyright-free pictorial archive series. This series is an marvelous and inexpensive source of graphics — something to consider if you need clip art for a publication of yours.

Feedback

We want to thank subscribers to the Analyst for their support, encouragement, kind remarks, and constructive suggestions.

We've had several requests for articles on specific subjects. String scanning seems to be a concern to many Icon programmers, and our current series on the subject is in response to this interest.

We've also been asked to present case studies of programs. This actually is hard to do. Most interesting programs are a bit bulky for the space available in a publication of this size and a detailed description takes a lot of room too. We'll give this one a try, however.

One subscriber asked for articles on SNOBOL4. We're reluctant to do this. This *is* a newsletter about Icon. We also doubt that a significant number of our subscribers are familiar with SNOBOL4.

If you suggest something that doesn't appear promptly, don't be surprised or discouraged — we try to stay several issues ahead of the publication schedule so that we don't have to scramble at the last minute to get an issue out. For example, the next three issues of the Analyst already are substantially complete.

But we are interested in what you'd like to see in the Analyst and we take all suggestions seriously, even if it sometimes takes us a while to do something about them. Please let us know what you think.

What's Coming Up

The focus on string scanning will continue in the next issue of the Analyst with two articles. One article will present some examples of scanning expressions and explain why they are written as they are. The other article will discuss pattern matching and the principles that underlie it.

And we have an article called "Gedanken Debugging".

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

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

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

(602) 621-8448

FAX: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

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

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona