The Icon Analyst

In-depth coverage of the Icon Programming Language

August 1990
Number 1

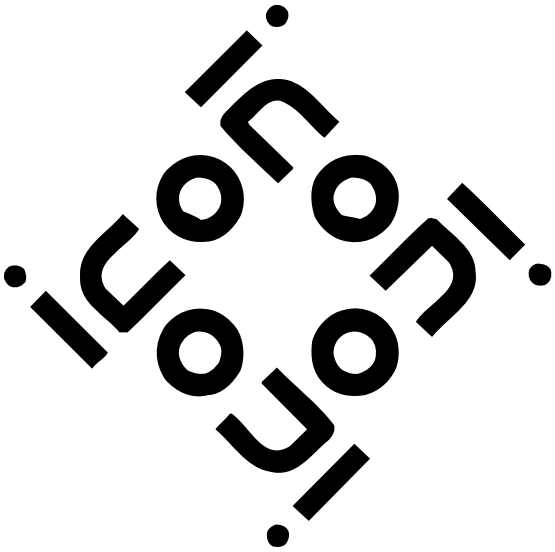
In this issue …

Launching The Icon Analyst

This is the inaugural issue of The Icon Analyst. The Analyst is devoted to technical aspects of the Icon programming language, and complements *The Icon Newsletter*, which features topical issues and user contributions.

The Icon Analyst contains articles with in-depth coverage of specific topics, programming tips, and so forth.

We want to make the Analyst useful to you. Your comments and suggestions are welcome. Just write, call, or send electronic mail. See page 6 for addresses and numbers.



Version 8 of Icon — An Overview

Version 8 of Icon has been released, and implementations for most computers are now available. What's Version 8 all about? If you're using an older version, should you update? To answer these questions, let's start with a review of what's new in Version 8.

Language Features

In the first place, there are no radically new features in Version 8. Most of the new features add functionality and provide refinements. For example, Version 8 has a set of functions for doing mathematical computations. These functions are similar to those you'll find in other programming languages — no surprises, but these new functions can be very handy if you need to compute a logarithm or the sine of an angle. They go toward making Icon useful for a wider variety of applications.

Version 8 also provides "keyboard functions" for doing direct input and output from a computer console. For example, getche() gets a character typed from the keyboard and echoes it on the screen. Keyboard functions are a virtual necessity for writing interactive screen-management applications. Their inclusion in Version 8 fills a gap that has frustrated many Icon programmers in the past.

It's worth noting that these functions are not supported on all systems. Typically, keyboard functions are supported on personal computers, but not on UNIX or on systems that evolved from the mainframe mentality.

Version 8 of Icon supports arithmetic on integers of arbitrarily large magnitude. While ordinary integer arithmetic

Version 8 Highlights

Math functions: sin(), cos(), … exp(), log(), …
Keyboard functions: getch(), getche(), kbhit()
Invocation with lists: p!L
Large-integer arithmetic
Variable access: name(v), variable(s)
Table keys: key(T)
Serial numbers for structures
Calling C functions from Icon and vice versa
Memory monitoring
Dynamic hashing

is performed as before on "machine integers", once an integer gets large enough to overflow, a "large integer" comes into existence. There is no limit to the size of large integers except the memory they require and the time it takes to perform computations on them.

Unless you're interested in number theory (one field of mathematics that's still accessible to amateurs), you may not see how large-integer arithmetic can be useful. But if you want to see all the digits in the largest known prime, here's your chance (as of this writing, it's $391,581 * 2^{216,193} - 1$ and has 65,087 decimal digits). If you don't need large integers, the nice thing is that they won't bother you. They don't affect performance until machine arithmetic would overflow. There's one reservation about this new feature: It involves a lot of code in the implementation — so much so that it is not included for personal computer systems that must work with a severely limited amount of memory.

A somewhat more esoteric capability in Version 8 is provided by functions that return the string names of variables and vice-versa. It's a bit difficult to imagine what these functions might be good for, although debugging comes to mind.

As mentioned above, Version 8 has many features that are best described as refinements or even cosmetic improvements. An example is the new keyword &letters whose value is a cset containing all the letters. It's no longer necessary to form the union of csets of upper- and lowercase letters. This isn't a big deal, but it will be welcomed by many programmers.

Every structure (list, set, table, and record) now has a unique serial number, which starts at one for each type and increases as more instances of the type are created. This serial number appears as part of the string image of a structure. Consequently, it's now possible to identify individual structures; that's very handy for debugging.

Another nicety of Version 8 is a way to invoke a function or procedure with an (Icon) list of arguments. For example, write!L writes the values in L. Formerly, it was necessary to know how many arguments were needed in every function call and write them out as part of the program. Now the number of arguments needn't be known until the function is called. This new method of providing arguments complements the ability to define a procedure with a variable number of arguments and makes certain kinds of programming techniques possible that could not be used before. Programmers from Lisp backgrounds will particularly appreciate this feature

Yet another feature that fills a gap is the function key(T), which generates the keys (entry values) in the table T. Until Version 8, the only way to do this was to sort the table into a list and pick out the keys from the list. Not only does the function do what's needed directly, but it also saves the time and space needed to sort the table. Incidentally, element generation from tables should have done this from the beginning. Generating the values instead of the keys was just a design mistake. But once a mistake like that is made, it's hard to fix it — programs use such features, even if the features are poorly designed.

One new feature of Version 8 that will be important to persons who want to expand Icon's computational repertoire is the ability to call "external" functions written in C (or another language with a C calling interface). The mechanism for doing this in Version 8 is primitive. Nonetheless it is easy, for example, to access functions in a graphics library. It is not necessary to recompile all of Icon to do this, although it is necessary to relink its run-time system. One potential problem is passing arguments. While there are existing methods for handling strings, integers, and floating-point numbers, figuring out how to pass structures (either Icon's or C's) is left to the user.

For the adventuresome, it's also possible to call an Icon program from C. It's even possible to call Icon from C, C from Icon, and so on, recursively. It's a bit hard to imagine a real application for this, however.

While it may sound exciting to have all of Icon's high-level computational capabilities available by calling Icon from C, there's a kicker. In order to be able to call an Icon program from C, you have to load the entire Icon run-time system along with your C program, not to mention the storage regions Icon allocates when it starts up. This is only practical on computers with a large amount of memory, and all the overhead may be unappealing even there.

Implementation Changes

The implementation of Icon contains some improvements that may be of more practical importance to some users than any of its new language features.

The most important improvement is "dynamic hashing" for sets and tables. This allows sets and tables to reorganize themselves as they grow in order to maintain good look-up performance. For programs that deal with really large sets and tables (such as in-memory word lists), performance is dramatically improved by dynamic hashing. It may make the difference between a practical application and an impractical one.

Structures also are somewhat smaller in Version 8. This improvement allows personal computer users to deal with larger amounts of data than they could before.

Memory Monitoring

In an entirely different category is Version 8's ability to produce a file detailing storage allocation and garbage collection as a program executes. While this is not the kind of thing most Icon programmers need or even would care to try to understand, it provides input for tools that can do everything from generating charts summarizing memory management to providing interactive color displays of Icon's allocated data regions. At present there are only a few such tools. The Icon Project provides one that produces color PostScript snapshots

of Icon's the data regions (it prints in black and white on garden-variety PostScript printers). Version 2.0 of ProIcon will have a tool for producing color PICT displays of Icon's data regions on the Macintosh II. More tools are sure to come — if you ever see a display, you'll understand why. See the article on page 7 for an introduction to memory monitoring.

User Impacts

Going to Version 8 of Icon from an earlier version will have almost no impact on most users. While no language change is totally transparent, most programs written for earlier versions of Icon run under Version 8 without modification. There's one reservation: Version 8 of Icon is larger than previous versions — this is the price of increased functionality. Users of personal computers with very limited amounts of memory may have trouble getting Version 8 to run with large Icon programs.

The New Book

The second edition of *The Icon Programming Language* appeared coincident with the release of Version 8. The second edition describes all the features of Version 8 of Icon. This should be welcomed by Icon programmers who, up to now, have had to deal with the out-of-date first edition and supplementary reports. The second edition is organized differently from the first edition and presents the important and interesting features of Icon first, before going on to the more mundane computational repertoire. For example, generators are described in Chapter 2 and string scanning is covered in Chapter 3. Teachers should welcome this approach, since it allows them to present the intellectually interesting part of Icon from the start. The new approach also simplifies subsequent examples and encourages good programming style. Other features of the second edition are more exercises (including some more challenging ones), additional reference material, a detailed discussion of running Icon programs, and several examples of large Icon programs.

The Icon Program Library

Along with Version 8, there is a new, larger version of the Icon program library — 63 programs and 48 sets of procedures. The new program library contains everything from the mundane to the esoteric; text utilities, games, tools for working with Icon programs, you name it. Specifics aside, the program library provides numerous examples of Icon programming techniques. It's a good way for an Icon novice to get started. For those interested in object-oriented programming, the library also includes Idol, an object-oriented version of Icon written in Icon.

The Bottom Line

If you haven't upgraded to Version 8, should you? Well, if you're using an earlier version, you're not having any troubles, and you don't see any new features you think you need, the motivation to upgrade probably is mostly a matter of staying current. (If you have a problem with Icon, the Icon Project probably will not be able to give much help unless you're running the latest version.) On the other hand, upgrading is inexpensive and relatively painless. You'll get some useful new features and improved performance, and you'll be in a position to follow new developments.

The Icon Programming Language
Second Edition

Ralph E. Griswold and Madge T. Griswold, Prentice Hall, 1990. 367 pages. $32. ISBN 0-13-447889-4.

Chapters:

1. Getting Started
2. Expressions
3. String Scanning
4. Csets and Strings
5. Arithmetic and Bit Operations
6. Structures
7. Expression Evaluation
8. Procedures and Variables
9. Co-Expressions
10. Data Types
11. Input and Output
12. Running an Icon Program
13. Programming with Generators
14. String Scanning and Pattern Matching
15. Using Structures
16. Mappings and Labelings
17. Programming with Strings and Structures

Appendices:

Syntax
Characters
Reference Manual
Error Messages
Implementations Differences
Sample Programs
Solutions to Selected Exercises

References

Index

Getting Started with Icon

This article is the first in a series designed to help programmers who are just starting to use Icon — or those who would like to use Icon but have been hesitant to dive in. These articles assume you have some programming experience. It's possible to start with Icon as a first programming language, but you'll need more preparation than is provided here. This series of articles assumes you are familiar with a programming language such as Pascal or C.

While it's easier to start out in Icon if you have experience with a language that has a similar syntax (like Pascal or

C), when you get around to writing real programs in Icon, you may find you have to "unlearn" some things to use Icon properly. The more experienced you are with another language, the more serious this "culture shock" is likely to be. These are semantic matters, however. We'll start with program structure and syntax.

Program Structure

An Icon program is composed of procedures that divide the computations the program performs into (supposedly) logical units. There is always a main procedure, which is where a program starts when you run it. The main procedure may call other procedures, and so on, to perform various tasks.

When starting out in Icon, it's best to first write a few programs that have only a single, main procedure before going on to more complicated programs. A simple example is

```
procedure main()
  write("Okay, let's get started!")
end
```

This is a procedure declaration. It gives the name of the procedure (main) and includes an expression that is evaluated when the program is executed. The parentheses after the procedure name are necessary; they're for parameters (more on this later). This program just writes Okay, let's get started!, as you'd expect.

If a procedure contains several expressions, they are evaluated in order, as in

```
procedure main()
  write("Okay, let's get started!")
  write("I'll write a bigger program soon.")
  write("But this is all for now.")
end
```

In case there are several procedures in a program, their declarations are written one after another. The order isn't important, but it's usually easier to read a program if the main procedure comes first. If there are many other procedures, it's good practice to group them logically or alphabetically so that it's easy to find them.

A procedure declaration cannot occur inside another. Procedure declarations cannot be nested and Icon does not support sub-procedures or block structure either. An example of a program is:

```
procedure main()
  hello()
  work()
  goodbye()
end

procedure hello()
  write("Okay, let's get started!")
end

procedure work()
  write("I'm not ready yet!")
end

procedure goodbye()
  write("But wait for next time!")
end
```

As suggested by this program, when a procedure is called, the expressions in it are evaluated and then it returns to the caller — when evaluation "flows off the end". More on this in a subsequent article.

Reserved Words and Identifiers

Every programming language has its own syntax — rules of grammar that determine what's correct and how programs are parsed.

Learning the syntax of a new programming language is always somewhat painful. Despite similarities of syntax, there are minor and sometimes major syntactic differences between different languages. Subtle differences between similar constructions often are the most troublesome.

Icon uses *reserved words* to identify important structural components of its syntax. Examples of reserved words are procedure and end, which are used to delimit procedure declarations as shown in the examples above. Icon also uses reserved words, such as if, then, and else, for control structures that determine how expressions are evaluated. You'll need to learn Icon's reserved words, since they identify important aspects of its syntax. This comes naturally as you learn the language.

Other "words" of your choice can be used for *identifiers* that are used to refer to values. For example, the expression

```
salutation := "Hello world"
```

assigns the string "Hello world" to the identifier salutation. Reserved words, however, cannot be used for identifiers. When you first start programming in Icon, you're likely to make a mistake such as

```
then := "I'm not ready yet!"
```

Icon's compiler will tell you that you've made a mistake, since then is a reserved word.

Without describing the syntax of identifiers in detail, it's worth noting one general rule in Icon: Upper- and lowercase letters are distinct and bear no relationship to each other as far as Icon is concerned. Consequently,

```
Then := "I'm not ready yet!"
```

is perfectly legal, since Then is not a reserved word.

If you're used to a programming language that doesn't distinguish between cases, you may make the opposite mistake in Icon:

```
PROCEDURE MAIN()
  WRITE("Hello world")
END
```

There are several problems here. PROCEDURE and END are not the same as procedure and end; Icon's compiler will

complain. Even if it didn't complain, MAIN is not the same as main, so this program wouldn't execute for lack of a main procedure. And even if it did execute, WRITE is not the same as write; WRITE is not a function and trying to call it would be an error too. The general rule is that all reserved words and function names are lowercase.

Icon's case distinction gives you freedom. You may find it handy, for example, to use uppercase or an initial uppercase letter to make it easy to distinguish identifiers that have special meaning to you.

Expressions

A procedure contains expressions. As mentioned earlier, these expressions are evaluated one after another in the order they appear in the procedure. This *sequential evaluation* can be interrupted temporarily by calling another procedure. Sequential evaluation also can be modified by *control structures*, as in

```
if i > j then i := j else j := i
```

which evaluates one of two expressions, depending on the relative magnitudes of i and j. Similarly, the loop

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

evaluates an expression repeatedly.

Unlike most (but not all) programming languages, Icon has no statements; all computations are performed by expression. The distinction, roughly speaking, is that statements control evaluation but do not produce values, while expressions produce values. In languages that have both expressions and statements, there are rules about where each can appear. Icon is free of these distinctions.

While it may appear strange at first,

```
if i > j then i := j else j := i
```

is an expression in Icon and produces a value (the value produced by the selected expression). For example, in Icon, you can write

```
write(if i > j then i := j else j := i)
```

which does the same thing as

```
write(i := j)
```

or

```
write(j := i)
```

depending on the relative magnitudes of i and j. Of course, unless you know Icon pretty well, you can only guess what these do.

Since expressions are more general than statements, you can use expressions just as you would use expressions in most other programming languages and not have to worry about it. Furthermore, you don't have to learn special rules about where expressions and statements can be used. In Icon, it's really simple: any expression is legal anywhere.

On the other hand, Icon's expression-based syntax sometimes lets you write things more compactly than you could in other programming languages. But such techniques are not part of "getting started".

Since expressions are evaluated in order (in the absence of control structures), something is needed to tell where one expression ends and another begins. Most programming languages use semicolons for this purpose. Icon does too. For example,

```
write("Hello."); write("Good–bye.")
```

consists of two expressions; the semicolon separates them. The semicolon is necessary; Icon's compiler objects to

```
write("Hello.") write("Good–bye.")
```

Icon provides semicolons for you automatically if you write expressions on separate lines, as in

```
write("Hello.")
write("Good–bye.")
```

In fact, it's never necessary to use a semicolon in an Icon program to separate expressions. You can always use separate lines. This technique also is good style; it generally makes programs easier to read.

But suppose you have a very long expression. While where is no limit on the length of a line in an Icon program, long lines are hard to read and may be wrapped around or truncated on terminals and printers.

Icon lets you write an expression on as many lines as you like — long expressions are automatically continued from line to line as necessary. For example,

```
count := cmp(1) –
  cmp(2) –
  cmp(3) –
  cmp(4)
```

is equivalent to

```
count := cmp(1) – cmp(2) – cmp(3) – cmp(4)
```

However, since Icon also allows expressions in sequence to be written on several lines without your having to provide the semicolons, the Icon compiler must be able to tell if lines in succession consist of separate expressions or a continued expression. The rule is this: If an expression ends on a line and the next line begins another expression, the expressions are separate. Otherwise, the second line is a continuation of the first.

This may sound so obvious that you wonder why it's said. The problem is ambiguity. For example,

```
count := cmp(1)
  – cmp(2)
  – cmp(3)
  – cmp(4)
```

is four separate expressions, not one long one. The difference between this example and the previous one is that, for example,

```
– cmp(2)
```

is a perfectly legal Icon expression. The operator – in prefix position produces the negative of cmp(2). Consequently in this example, each line contains a complete expression, even if the result is not as intended. However,

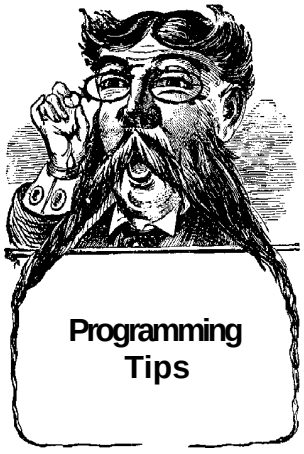
```
count := cmp(1) –
```

and subsequent lines in the earlier example are not complete expressions, so the expression is continued until a complete expression is encountered.

You may think you'll need an expert knowledge of Icon's syntax and constant diligence to cope with this problem. Actually there's a simple rule: It you want an expression to be continued on several lines, all you need to do is be sure every line except the last one does not end a complete expression. That's actually easy to do. Since Icon has no suffix operators, any line ending with an operator cannot end an expression.

Next Time

In the next article we'll discuss more about syntax, including pitfalls and how to avoid them. We'll also discuss the basic aspects of expression evaluation in Icon: success, failure, and generation.



This regular feature of the Analyst is devoted to aspects of programming in Icon that may not be obvious — things you can do to improve your programs or make them run faster.

For example, to improve performance in inserting new elements in a table, use

```
insert(T,x,y)
```

instead of

```
T[x]:= y
```

Both do the same thing, but the latter is considerably slower.

There are several reasons for this difference in performance, but the primary one has to do with the way expressions are evaluated in Icon. Consider the general case of assignment to a subscripted table:

```
T[x] := expr
```

where *expr* is some expression, perhaps a complicated one. Evaluation of the left side of the assignment requires that x be looked up in T. Suppose x is not in T. The evaluation of T[x] detects this and prepares for the insertion of a new element into T. The expression on the right side of the assignment is evaluated next. Suppose it inserts an element into T. It could even insert x into T. While this may appear unlikely, it is possible, and Icon has no way of knowing that it will not happen. Consequently, when the assignment finally is performed, x must be looked up in T again — just in case.

On the other hand,

```
insert(T,x,expr)
```

presents no such problem, since *expr* is evaluated before insert() is called and there's no possibility of a side effect such as the one described above. One look-up will do in this case. For similar reasons, it's faster to use member(T,x) than to check T[x] for (say) the default value.

Of course the implementation of Icon could be smarter about table subscripting, but it isn't and if you want to speed up a program that does a lot of table subscripting, insert() and member() are the way to do it — not pretty, perhaps, but certainly faster.

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

Memory Monitoring

Icon has many kinds of data. Much of the power of Icon comes from all the things you can do with different kinds of data. Furthermore, objects of these types are created as they are needed during program execution — when you write an Icon program, you don't have to know how many data objects you will need or even how big they will be.

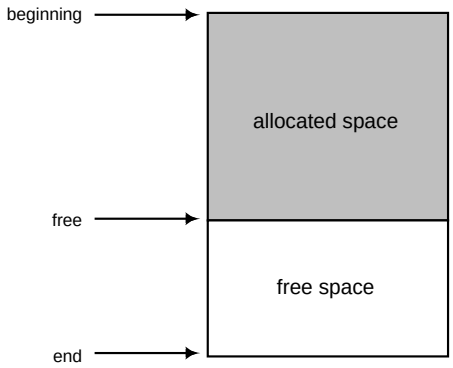
Allocation and Garbage Collection

Space for such objects is allocated, and unused objects are "garbage collected" when more space is needed. All this is automatic — you don't have to account for it when you write a program. Nevertheless, objects that are created during program execution and the memory space they occupy can be significant factors in program performance.

The situation is complicated by the fact that Icon often provides several different ways of performing an operation using different kinds of data. You may, for example, have a choice between using a list, a set, or a table to keep track of a collection of values. One choice may be easier to program than another, but you may also wonder about the relative amount of memory space used by different methods. Normally, you'd probably prefer not to worry about this, but sometimes the choice of a data type makes the difference between a program that works and one that doesn't — for lack of space. You might also just be curious about how Icon allocates and collects objects during program execution. Memory management in Icon is described in detail in Reference 1. Here's a brief overview.

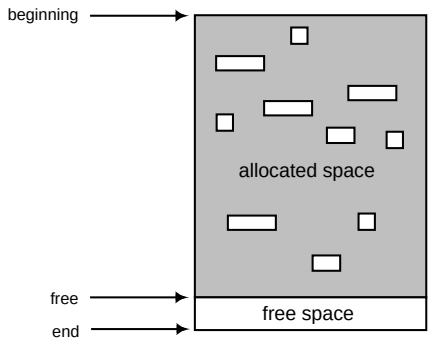
Icon allocates space for objects that are created during program execution in two main regions: a string region and a block region. The string region consists of characters, while the block region contains structures and related objects. Allocation in the string region is in terms of bytes, while allocation in the block region is in terms of "words". A word is four bytes for most implementations of Icon. Some implementations of Icon also have a "static" region, which is not described here.

Allocation proceeds in the same manner in both main regions. The regions initially are empty and bounded by pointers. As space is needed, it is provided starting at the beginning of the region. A "free" pointer is incremented to mark the boundary between *allocated space* and *free space*:

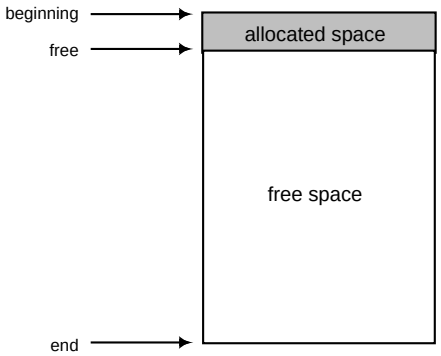


If there is not enough available free space to satisfy an allocation request, a garbage collection is performed to reclaim space occupied by objects that are no longer needed.

The garbage collection process is fairly complicated, since it is necessary to locate all objects that may be needed for subsequent program execution. Objects that need to be saved typically are scattered throughout the string and block regions:



Once the objects to be saved are identified ("marked"), they are relocated toward the beginning of their region, compressing the allocated space and making more free space available so that allocation can proceed:



Allocation Histories

Version 8 of Icon provides a tool called memory monitoring that provides a way of finding out how memory space is used. Memory monitoring ordinarily is disabled and you'd not know it was there. It is enabled by setting the value of the environment variable MEMMON to the name of a file before you run your program. If this variable is set, memory management information is written to the file as your program runs. This *allocation history file* contains a detailed record of all the storage allocation that your program performs, as well as what goes on during garbage collection. The box on the next page shows a typical allocation history file. An encoding of the details of memory management follows header information.

After execution, such an allocation history file can be used in a variety of ways. An allocation history file, by its nature, is too detailed to use directly. But there are several tools for processing allocation history files, ranging from simple summaries of storage management to animated color visualizations of memory management in action.

A Typical Allocation History File

```
##  Icon MemMon output
#
#   program: concord
#   date:    Tue Apr 17 14:19:02 1990

4< 131860:0/0 138240:0/65000 204800:0/65000
0"
>
= 131860:0/0 138240:0/65000 204800:0/65000
5L23l16T10h0"6"0"6"0"6"0"6"0"6"0"6"0"6"0"6"0"6"0"6"0"6"60"6"60"7eL
9l54+7%rteLl75+%rteLl96+%rteLl117
+%rteLl138+%rteLl159+%rt0"6"0"6"0"6"68"
6"68"eLl180+%rteLl201+%rteLl222+%rteLl243+%rteLl264+%rt23leL9l
Ll329+%rt66"6"66"eLl350+%rteLl371+%rteLl392+%rteLl413+%rteLl
eL9l478+%rt23leL9l522+%rt64"6"64"eLl543+%rteLl564+%rteLl585+%rteLl
eLl627+%rteLl648+%rteLl669+%rt67"6"67"eLl690+%rteLl711+%rteLl
l753+%rteLl774+%rteLl795+%rteLl816+%rteLl837+%rteLl858+%rt69"6"69"
eLl900+%rteLl921+%rteLl942+%rteLl963+%rtheLl994+%rteLl1015+%rteLl
64"6"64"23leL9l1080+%rteLl1101+%rt23lll69"6"69"eL9l1191+%rteLl
eLl1233+%rteLl1254+%rteLl1275+%rteLl1296+%rt68"6"68"eLl1317+%rteLl
eLl1359+%rteLl1380+%rteLl1401+%rteLl1422+%rt0"6"68"6"68"eLl
1464+%rt23leL9l1508+%rteLl1529+%rt23leL9l1573+%rt76"6"76"eLl
l1615+%rteLl1636+%rteLl1657+%rteLl1678+%rteLl1699+%rt68"6"68"eLl
eLl1741+%rteLl1762+%rteLl1783+%rt23leL9l1827+%rt64"6"64"eLl
lleL9l1938+%rteLl1959+%rteLl1980+%rteLl2001+%rt23leL9l2045+%rt69"6"
eLl2066+%rt18h23llleL9l2174+%rteLl2195+%rt68"6"68"eLl2216+%rt23llleL
eLl2304+%rteLl2325+%rteLl2346+%rt65"6"65"eLl2367+%rt23lleL9l
6"16"eLl2455+%rt23l0"6"68"6"68"eL9l2499+%rteLl2520+%rteLl2541+%rteL
"6"68"eLl2583+%rteLl2604+%rteLl2625+%rteLl2646+%rt23lleL9l2713+%rte
eLl2755+%rt63"6"63"eLl2776+%rteLl2797+%rteLl2818+%rteLl2839+%rteLl
66"6"66"eLl2881+%rteLl2902+%rteLl2923+%rteLl2944+%rteLl2965+%rteLl
eLl3007+%rt67"6"67"23leL9l3051+%rteLl3072+%rteLl3093+%rteLl
67"23leL9l3158+%rteLl3179+%rteLl3200+%rt23leL9l3244+%rteLl
34"6"34"eLl3307+%rteLl3328+%rteLl3349+%rt0"6"68"6"68"eLl3370+%rt23l
3414+%rteLl3435+%rt67"6"67"eLl3456+%rt23llllll"6"67"llll52"6"52"llll0"
6"77"6"77"leL9l3822+%rteLl3843+%rteLl3864+%rteLl3885+%rteLl
6"64"eLl3927+%rt23leL9l3971+%rt23leL9l4015+%rt23l17"6"17"ll0"6"68"6"
eL9l4105+%rteLl4126+%rteLl4147+%rt67"6"67"23leL9l4191+%rteLl
l4233+%rteLl4254+%rteLl4275+%rt69"6"69"eLl4296+%rteLl4317+%rteLl
23llleL9l4428+%rt65"6"65"eLl4449+%rteLl4470+%rteLl4491+%rt68"6"68""
eLl4512+%rteLl4533+%rt23leL9l4577+%rteLl4598+%rt0"6"0"6"0"6"0"6"0"6"
6"69"6"69"eLl4619+%rteLl4640+%rt0"6"0"6"0"6"0"6"0"6"0"6"58"6"58"eLl
34heLl4716+%rteLl4737+%rteLl4758+%rt30"6"30"eLl4779+%rt"6"30"eLl
26"6"26"0"6"0"6"66"6"66"eLl4821+%rt23ll"6"66"leL9l4911+%rteLl4932+%rt
eL9l4976+%rt64"6"64"eLl4997+%rteLl5018+%rt23leL9l5062+%rt68"6"68"
9l5106+%rteLl5127+%rteLl5148+%rteLl5169+%rteLl5190+%rt66"6"66"eLl
eLl5232+%rt23leL9l5276+%rt23l65"6"65"eL9l5320+%rteLl5341+%rt23leL9l
68"6"68"eLl5406+%rteLl5427+%rteLl5448+%rt"6"68"eLl5469+%rt23leL9l
eLl5534+%rt23leL9l5578+%rteLl5599+%rt69"6"69"eLl5620+%rteLl
5662+%rteLl5683+%rteLl5704+%rt23ll"6"69"31leL9l5802+%rt23leL9l
68"6"68"eLl5867+%rteLl5888+%rteLl5909+%rt"6"68"eLl5930+%rt23llll65"
67"6"67"eL9l6043+%rteLl6064+%rt23lleL9l6131+%rt65"6"65"eLl6152+%rte
23lleL9l6240+%rt66"6"66"eLl6261+%rt23leL9l6305+%rt23leL9l6349+%rt
64"eLl6370+%rteLl6391+%rteLl6412+%rt23leL9l6456+%rt67"6"67"23llleL
34"6"34"eLl6590+%rt0"6"67"6"67"eLl6611+%rteLl6632+%rteLl6653+%rteLl
eLl6695+%rt65"6"65"eLl6716+%rteLl6737+%rt69"6"69"eLl6758+%rt23leL9l
23llleL9l6892+%rt23l92"6"92"eL9l6936+%rteLl6957+%rteLl6978+%rteLl
eLl7020+%rt71"6"71"eLl7041+%rteLl7062+%rteLl7083+%rteLl7104+%rt23l
69"lleL9l7194+%rt64"6"64"23lleL9l7261+%rteLl7282+%rt43"6"43"0"6"64"6"
23ll62"6"62"lleL9l7395+%rt22"6"22"eLl7416+%rt0"6"65"6"65"23l66"6"66"
23l63"6"63"eL9l7526+%rt23lll69"6"69"31leL9l7647+%rteLl7668+%rt23l88"
```

and so on …

Allocation Summaries

Let's start with something simple — a summary of allocation. Consider the following simple program for counting the number of times each character occurs in a file:

```
procedure main()
   ccount := table(0)                  # table of characters
   while ccount[reads()] +:= 1         # count characters
   ccount := sort(ccount,3)            # sort them
   rwidth := 0                         # count width
   every rwidth <:= *!ccount
   while write(left(image(get(ccount)),10),
      right(get(ccount),rwidth))       # output results
end
```

Here's a summary of allocation obtained by processing the allocation history file that results when this program is run with an input file containing 30,656 characters:

type	number	bytes	average	% bytes
string	31911	35628	1.12	75.508
list header	2	40	20.00	0.084
table header	1	64	64.00	0.135
table element	251	7028	28.00	14.894
hash block	4	288	72.00	0.610
list element	2	4136	2068.00	8.765
total:	32171	47184	1.47	

The first line shows the allocation in the string region. The remaining lines show the objects allocated in the block region. You need to know something about how Icon represents its data structures to interpret these figures. For example, every list has a header, which links to blocks containing list elements. A table also has a header, as well as hash blocks and table elements.

If you look carefully at the summary above, you may have a few questions. For example, two list headers were allocated, which means the program created two lists. One is obvious — the one produced by sorting the table. The other list is the one for the argument to main(). Even though main() in this program has no argument, the list is created anyway — it's not easy to find out if main() needs an argument at the time the command line is processed for arguments. So there's always one list allocated in addition to any the program itself may create.

It's worth looking at the rest of the summary to see if there is anything surprising. Most storage is allocated for strings, which makes sense, since all of the input file has to be read (Icon allocates string space for any data it reads). The average of 1.12 for the length of allocated strings is plausible. There are two sources of strings in the program: the one-character strings produced by reads() and the strings produced by image(s). Most of the imaged strings are three characters long (the character itself plus the surrounding

quotes). The images are longer for characters that are not "printable" and hence are represented by escape sequences.

The 64 bytes for the table header tell you something — any table is going to take at least this much space. There is one hash block initially, and more are added as elements are added to the table. The details of this process are described in Reference 2.

Every table element occupies 28 bytes. There evidently are 251 different characters (one table element for each) in the input file for this run (it was a "binary" file). Since there are only 256 possible characters, the space for the table and its associated components cannot be much bigger than in this example.

The maximum amount of space required by this program is, in fact, the sum of the size of the input file plus a constant. You might try figuring out what this constant is — and check it by testing the program with an input file that contains just one each of each of the 256 characters. The test file is easy to produce:

```
procedure main()
   every writes(!&cset)
end
```

Space is not an important matter in this program (string space for data read in is garbage-collected automatically). But you might wonder if the program could be written to use less space than in the example. One possibility is to avoid the table altogether by using the ordinal values of the characters to index a list of counts:

```
procedure main()
   ccount := list(256,0)          # list of counts
   while ccount[ord(reads()) + 1] +:= 1
   rwidth := 0
   every rwidth <:= *!ccount
   every i := 0 to 255 do
      if (c := ccount[i + 1]) > 0 then
         write(left(image(char(i)),10),right(c,rwidth))
end
```

Of course, this program is not "Iconish" in style. It's like the kind of program that C programmers often produce when first introduced to Icon. But how much storage does it use? Here's the summary:

type	number	bytes	average	% bytes
string	32162	35879	1.12	94.202
list header	2	40	20.00	0.105
list element	2	2168	1084.00	5.692
total:	32166	38087	1.18	

Although the total space is not much less than for the first version of the program, the amount of non-string (block) space is much less: 2,208 bytes as opposed to 11,556 bytes.

Comparing this summary to the first one might tell you something about a detail of the implementation of Icon: char(i) allocates a one-character string. It doesn't have to be implemented this way; it could use a static array of 256 characters.

But why is less space allocated for list elements in the second version of the program? That's not a quirk of the implementation. In the first version of the program, the size of the list produced by sorting is twice the number of different characters: one element each for the key and its value. In the second version of the program, the size of the list is just 256.

All this preoccupation with storage allocation does not deal with a question that is probably more important for this program: Which version is faster?

The first version is faster, by about 23%. The reason presumably is the extra computation (ord() and char()) in the second version. It's encouraging that the "Iconish" method is faster, at least.

Challenge — can you write a version of this program that is faster than the first one given here? Or one that uses less space than the second? How about one that doesn't use a list at all (while still producing its output in collating order)?

If you have Version 8 of the Icon program library, you can use the program memsum.icn there to get summaries similar to the ones above. This program also can produce tab-separated data for use in spreadsheets and charting programs.

The use of memsum is illustrated by the following commands for obtaining an allocation history file for the program tablc.icn on a UNIX BSD system (the details are different for other systems):

```
setenv MEMMON tablc.mon
tablc <tablc.dat
unsetenv MEMMON
memsum <tablc.mon >tablc.sum
```

It's essential to remove the setting for MEMMON before running memsum. Otherwise, since memsum is an Icon program, an allocation history file for it would be produced, overwriting the one for tablc. (If you ever get a summary showing all zero entries, this is the probable cause.)

Such summaries give an interesting overview of all the storage allocation in an Icon program, but they use only a small portion of the information in an allocation history file. As mentioned earlier, an allocation history file is too detailed to be understandable as it stands. There are, however, several other ways of using the data from an allocation history file. One possibility is a visualization of memory management.

Several such visualization programs exist. The simplest of these produces "snapshots" of Icon's storage regions at critical times, showing every allocated object according to its size and position in memory. Objects of different types are displayed in different colors to make them more readily identifiable (on black-and-white devices, shades of grays and patterns are used to achieve some of the same effects). More-

sophisticated visualization programs produce animated output, showing each object as it is allocated.

These visualizations provide much more insight into Icon's memory management than any number of summaries or charts. We'll describe visualization in more detail in the next issue of the Analyst.

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.

Benchmarking Icon Expressions

With this article, we're starting a series on writing efficient programs in Icon. This series will cover many aspects of programming in Icon. It starts here with a discussion of some basic concerns and then goes on to show how to measure expression evaluation, so that you can answer some of the questions you may have about Icon's performance.

Most lower-level programming languages mirror the architecture of the computers on which they run. It's fairly easy, for example, to relate code written in FORTRAN to the machine instructions that execute it. A programming language like Icon, however, has many features that do not have obvious images in machine instructions. Even Icon operations that look simple and "close to the machine" frequently are more complicated than they appear because of Icon's automatic storage management and type conversion.

Experience has shown that Icon programmers often have misconceptions about the efficiency of Icon expressions, sometimes underestimating their performance and sometimes overestimating. Programmers tend to expect expressions that appear to be far from machine instructions to be slower than they are, while overlooking hidden costs in operations that appear simple. Most Icon programmers, for example, expect `map(s1,s2,s3)` to be slower than it is and they mistakenly believe the activation of a co-expression is significantly slower than a procedure call. Even programmers who are familiar with the implementation of Icon often are mistaken about efficiency issues.

Efficient programming in Icon is further complicated by the richness of Icon's computational repertoire, which sometimes provides several different ways of performing the same computational task.

Analytic techniques, based on detailed knowledge of the implementation of Icon, can give average or worst-case figures for operations like set and table look-up. Such analytic descriptions of performance tend not to be very helpful because of the wide variation in data encountered in practice and the interaction of factors such as storage allocation and garbage collection.

Empirical results, obtained by actually timing different kinds of operations, often are more helpful than analytic results in providing guidelines for efficient programming techniques. Such measurements also sometimes produce unexpected results that dispel misconceptions (and even show problems in the implementation itself).

While it's easy to time a complete program, a finer level of detail is needed to answer questions about alternative programming techniques and to identify sources of inefficiency. It's not hard to time a single expression in a loop to get an approximation to the time it will take to evaluate it in the context of a real program. For example,

```
procedure main()
  itime := &time
  every 1 to 10000 do {
    abs(–1.0)
    }
  write(real(&time – itime) / 10000)
end
```

evaluates `abs(–1.0)` 10,000 times and writes the average time for an evaluation. The number 10,000 is a bit arbitrary, but the number of iterations should be large enough to overcome anomalies resulting from the discrete nature of the computer clocks. Most computer clocks "tick" 60 times a second, although some, such as the one for MS-DOS, change only once a second.

This simple program does not allow for the overhead of the loop, which may be as much or more than the time it takes to evaluate `abs(–1.0)`. There are many other issues, such as the effects of external factors like the load in a multi-tasking environment. Furthermore, if you want to time a lot of expressions, modifying the program, keeping track of information like the expression that was measured, the version of Icon, the date of the run, the system on which the timing was done, and so forth become daunting chores. The obvious solution is to write a program to take care of these menial tasks automatically.

A Program to Produce Measurement Programs

What's more natural than to use an Icon program that produces Icon programs to do the measurements, record the information, and so on?

For naive timings, a program that produces loops such as the one above is all that's needed. However, something needs to be done about compensating for loop overhead. This turns out to be a bit tricky. It's not good enough to compute loop overhead by timing an empty expression, as in

```
every 1 to 10000 do { }
```

The problem is that the Icon compiler discards the empty do clause, and hence some code that is present when timing a non-empty expression. A better approximation is

```
every 1 to 10000 do {
  &null
  }
```

Although &null is about as fast as any expression, the time it takes to evaluate it is significant compared with other simple expressions.

If you look at the code generated by the Icon compiler, you'll see that conjunction introduces very little overhead, so it might suffice to use

```
every 1 to 10000 do {
  &null & abs(–1.0)
  }
```

and subtract the overhead for &null. This still leaves the (small) overhead for conjunction. To get rid of that, the difference in times for

```
&null & &null
```

and

```
&null
```

does the trick.

Enough of these complexities. Here's a simple program to produce expression measurement programs:

```
procedure main()
  write("procedure main()")
  write("   _Itime := &time")
  write("   every 1 to 10000 do { &null }")
  write("   _Over := real(&time – _Itime) / 10000")
  write("   _Itime := &time")
  write("   every 1 to 10000 do { &null & &null }")
  write("   _Over := real(&time – _Itime) / _
    10000 – _Over")
  while line := read(input) do  {
    write("   write(",image(line),")")
    write("   _Itime := &time")
    write("   every 1 to 10000 do {")
    write("      &null & ", line)
    write("      }")
    write("   write((real(_Itime) / 10000) – _
      _Over,\" ms.\")")
    }
  write("end")
end
```

This program, empg (for "expression measurement program generators"), reads expressions from standard input. When the resulting program is run, the timing is performed and relevant information is written.

For example, if the file abs.exp contains the following lines,

```
abs(1.0)
abs(–1.0)
```

then

```
iconx empg <abs.exp
```

produces the program

```
procedure main()
  _Itime := &time
  every 1 to 10000 do { &null }
  _Over := real(&time – _Itime) / 10000
  _Itime := &time
  every 1 to 10000 do { &null & &null }
  _Over := real(&time – _Itime) / 10000 – _Over
  write("abs(1.0)")
  _Itime := &time
  every 1 to 10000 do {
    &null & abs(1.0)
    }
  write((real(_Itime) / 10000) – _Over " ms.")
  write("abs(–1.0)")
  _Itime := &time
  every 1 to 10000 do {
    &null & abs(–1.0)
    }
  write((real(_Itime) / 10000) – _Over, " ms.")
end
```

The identifiers in the program begin with underscores and capital letters to minimize the possibility of name collisions with expressions that are measured.

When this program is then run, the output looks something like this:

```
abs(1.0)
1.0076 ms.
abs(–1.0)
2.4246 ms.
```

The version of empg given above is primitive and is just intended to illustrate the idea. Lots more can be done, such as reporting environmental information and providing a way to specify the number of loop iterations. It's also clear that just timing all the expressions in a file is not enough. To time table lookup, for example, it's necessary to create the table outside of the timing loop. Although the table could be created in a separate timed expression, this is unnecessarily time consuming. Furthermore, to measure expressions that call procedures and reference records, it's necessary to have some way of getting declarations into the measurement program.

Thus, like many problems that start with a simple idea and grow into elaborate tools, empg can be made more sophisticated and a simple language can be designed for its input. The version of empg that follows, which by no means exhausts the possibilities, interprets an input line according to its first character, as follows:

The rest of the line is a comment to be written by the timing program.

: The rest of the line is an expression that is evaluated only once.

$ The rest of the line is part of a declaration and is appended to the end of the timing program.

The changes in empg are:

```
decls := [ ]                    # list for declarations
            :
            .

while line := read() do
  case line[1] of {
    ":": {                      # evaluate, not time
      write("   ",line[2:0])
      write("   write(",image(line[2:0]),")")
      }
    "$": {                      # line of declaration
      put(decls,line[2:0])
      write("   write(",image(line[2:0]),")")
      }
    "#":                        # comment
      write("   write(",image(line),")")
    default: {                  # time in a loop
      write("   write(",image(line),")")
      write("   _Itime := &time")
      write("   every 1 to 10000 do {")
      write("      &null & ", line)
      write("      }")
      write("   write((real(_Itime) / 10000)
        – _Over,\" ms.\")")
      }
    }
            :
            .
every write(!decls)             # write declarations
```

For example, the output for an expression file containing

```
:s := blank := " "
s ||:= blank
```

is

```
s := blank := " "
write("s := blank := \" \"")
write("s ||:= blank")
_Itime := &time
every 1 to 10000 do {
  &null & s ||:= blank
  }
write((real(_Itime) / 1000) – _Over, " ms.")
```

We've swept a lot of things under the rug here. The big one is storage management. We'll discuss this in a subsequent article.

In the meantime, if you have Version 8 of the Icon program library, you'll find a full-blown version of empg in it. If you don't have the program library, you can write your own, based on the examples here.

Try measuring some expressions and see if the results are what you expect. To test your intuition, guess which of the following expressions is fastest:

```
every !&digits
every !"0123456789"
every "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Now measure them and see.

What's Coming Up

In the next issue of The Icon Analyst, we'll continue the articles on memory monitoring and benchmarking Icon expressions. We'll also continue the series for beginning Icon programmers with an article on the basics of expression evaluation.

As with any programming language, there are some things you can write that look perfectly plausible but that don't do what you expect. We'll list some of these syntactic pitfalls in Icon in the next issue.

We'll have a couple more programming tips and start a new feature on things Icon "wizards" do.

Looking farther down the line, we plan a series on string scanning, and articles on program readability and writing portable Icon programs.

If you have a topic you'd like to see covered in The Icon Analyst, let us know. We can be reached by various means; see the box on Page 6.

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)