The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1992
Number 11

In this issue …

Data Representation: A Case Study

One of the joys of Icon is that it provides a wealth of data types and allows many ways of representing data in a program.

Very often the data representations you choose for a particular program have a significant effect on how you write the program. Although the algorithms you use probably would be the same whether you encode a tree as a string or encode it using records, the programming techniques are likely to be somewhat different. Perhaps more important, the resources the program requires may be quite different for different data representations.

While efficiency may not be the first concern in the design of a program, it often becomes an important consideration as the program matures. Memory utilization is usually the more crucial issue, since a program may not run at all if there is not enough memory available. This problem is particularly critical for personal computers in general and for MS-DOS in particular.

This article presents a case study of a program that produces a simple concordance — a listing of the lines in which the words of a text appear. The program is interesting because the choice of data representation affects the programming techniques only slightly, but it has a dramatic impact on its memory requirements. Because the choice of data representation doesn't affect programming techniques significantly, it is possible to compare different data representations easily.

The Concordance Program

The concordance program is basically quite simple:

1. A line of text is read in and written out with a line number.

2. Each word of the line is stored with its associated line number.

3. Steps 1 and 2 are repeated until all lines are read.

4. The word and line-number information then is organized and each word is printed out with a list of line numbers in which it occurs.

The program specification additionally requires that even if a word appears more than once on a line, the line number is listed only once. This turns out to be a crucial issue in the data representation.

A few other points about the program are worth mentioning:

• The definition of a word is naive — just a string of letters.

• Upper- and lowercase letters are considered to be equivalent.

• Words less than three characters long are omitted from the concordance.

An example of output from the concordance program is shown in the box below.

```
1        On the Future!–how it tells
2        Of the rapture that impells
3       To the swinging and the ringing
4        Of the bells, bells, bells–
5      Of the bells, bells, bells, bells,
6             Bells, bells, bells–
7   To the rhyming and the chiming of the bells!

and          3, 7
bells        4, 5, 6, 7
chiming      7
future       1
how          1
impells      2
rapture      2
rhyming      7
ringing      3
swinging     3
tells        1
that         2
the          1, 2, 3, 4, 5, 7
```

Output of the Concordance Program

```
#  Concordance program using a table of tables
#
global uses, lineno, width, min_size

procedure main(args)
   local word, line

   width := 15                  # width of word field
   min_size := 3                # smallest word to cite
   uses := table()
   lineno := 0

   every tabulate(words())      # tabulate the words

   output()                     # print the citations

end

#  Add line number to citations for word
#
procedure tabulate(word)

   /uses[word] := table()
   uses[word][lineno] := 1
   return

end

#  Generate words
#
procedure words()
   local s, line

   while line := read() do {
      lineno +:= 1
      write(right(lineno,6), "  ", line)
      map(line) ? while tab(upto(&letters)) do {
         s := tab(many(&letters))
         if *s >= min_size then suspend s
         }
      }

end

#  Print the results
#
procedure output()
   local word, line, numbers

   write()

   uses := sort(uses,3)         # sort citations
   while word := get(uses) do {
      line := ""
      numbers := sort(get(uses), 3)
      while line ||:= get(numbers) || ", " do
         get(numbers)           # skip marking value
      write(left(word,width), line ? tab(–2))
      }

end
```

A Concordance Program

A Table of Tables

When we wrote the first version of this concordance program several years ago, we kept the words in a table (a natural choice at the time). The question was then how to keep track of the line numbers for each word. We used tables for this also, with a table of line numbers for each word. Thus, the words and line numbers were kept in a table of tables.

The program itself consists of a loop in which the lines of text are read and written while the structures for the words and line numbers are built. The complete program is shown in the box at the left.

The procedure tabulate() is of particular interest, and the procedure output() is also relevant in the discussion here:

```
procedure tabulate(word)

   /uses[word] := table()
   uses[word][lineno] := 1
   return

end

procedure output()
   local word, line, numbers

   write()

   uses := sort(uses, 3)
   while word := get(uses) do {
      line := ""
      numbers := sort(get(uses), 3)
      while line ||:= get(numbers) || ", " do
         get(numbers)
      write(left(word,width), line ? tab(–2))
      }

end
```

A Table of Tables

In the procedure tabulate(), a check is first made to see if word is already in the table uses. If it isn't, it is assigned an empty table to hold its line numbers. Then the line number is added to the table for word by assigning 1 to it (any non-null value would do). Note that if the line number is already in the table, this assignment has no affect; duplicate line numbers are eliminated automatically.

In output(), uses is sorted to produce a list that consists of each word followed by the table of its line numbers. Then a line is built up for each word with the numbers of the lines in which it appears. The line is written with the trailing comma and blank removed by string scanning. (You might try your hand at seeing if the concatenation can be avoided by just

writing the data as it is processed, as described in recent issues of the Analyst.)

A Table of Sets

If you're an old hand at Icon, the use of a table of tables for keeping track of the words and line numbers probably seems very natural. If you're new to Icon, you may wonder why a table of sets wasn't used. The reason is simple: The original program was written before Icon had sets. Sets not only handle the "membership" issue with line numbers more naturally than tables, but sets also don't require as much memory as tables.

It doesn't take much to convert the program to use sets instead of tables. Only the procedures tabulate() and output() need changing:

```
procedure tabulate(word)
   /uses[word] := set()
   insert(uses[word], lineno)
   return
end
procedure output()
   local word, line, numbers
   write()
   uses := sort(uses, 3)
   while word := get(uses) do {
      line := ""
      numbers := sort(get(uses))
      while line ||:= get(numbers) || ", "
      write(left(word,width), line ? tab(–2))
      }
end
```

A Table of Sets

The two tabulate() procedures are even more similar if the one for tables uses insert():

```
/uses[word] := set()
insert(uses[word], lineno,1)
```

And, as you can see, the procedure output() for sets is somewhat simpler than it is for tables.

The issue might have ended here except for the fact that both a table of tables and a table of sets take a lot of memory. These two versions of the concordance program run out of memory very quickly when run under MS-DOS, for example.

A Table of Lists

Using sets to keep track of the line numbers and automatically eliminate duplicate ones is handy but hardly necessary. Lists can be used instead of sets, giving a table-of-lists representation.

Since the line numbers never decrease as words are processed, checking the last line number put onto the end of a list is all that's needed to avoid duplicates.The procedure tabulate() needs changing accordingly:

```
procedure tabulate(word)
   if /uses[word] := [lineno] then return
   if uses[word][–1] ~= lineno
      then put(uses[word], lineno)
   return
end
```

A Table of Lists

In addition, since the lists of line numbers are already sorted, it is not necessary to sort them in output():

```
numbers := get(uses)
```

is all that's needed.

With the change from sets to lists, the program requires considerably less memory than before and hence can handle larger texts that the previous versions when run on a platform with a limited amount of memory.

A Table of Strings

Having gone this far, we wondered if there was a less memory-intensive way to represent the data.

This led us to consider other data representations. It's hard to imagine keeping track of the words in a structure other than a table. There has to be way of finding the words. Icon does this efficiently with tables and trying to program something else is a big job that might come out badly.

On the other hand, it's clearly possible to build up the lists of line numbers "on the fly" and avoid using the memory used by the other versions to hold the line numbers until after all the input is read.

We'd been avoiding this approach, since we thought it would be awkward to detect duplicate line numbers in strings. It's actually not that bad when you get down to it, although we admit we didn't get it right on the first few tries and it took some work to get it to a point where it is now.

For a table of strings, uses now needs to have the empty string as the default value:

```
uses := table("")
```

Aside from this, the only changes to the program are again in tabulate() and output():

```
procedure tabulate(word)
   if uses[word][–2 –: *lineno] == lineno then
      return
   else {
      uses[word] ||:= lineno || ", "
      return
      }
end

procedure output()
   local word

   write()

   uses := sort(uses,3)
   while word := get(uses) do
      write(left(word, width), get(uses) ? tab(–2))
end
```

A Table of Strings

You might try to see if you can handle the problem of duplicate line numbers using string scanning. We haven't been able to come up with a solution that isn't tortuous.

Performance Comparisons

It's easy to make meaningful comparisons among the four programs, because the differences between them are minor and isolated.

We ran the programs under Version 8.0 of Icon with a 524-line input text containing 1730 words, 324 of which are different, and comprising a total of 18,354 characters. Here are the figures for memory utilization, in bytes:

version	tables	sets	lists	strings
string allocation	139,954	139,954	139,954	140,173
block allocation	141,036	110,771	71,180	51,512
total allocation	280,990	250,726	211,134	191,685
block region size	116,224	94,208	81,088	20,224

The block region sizes are the minimums required for the programs to run to completion.

As expected, the amount of memory allocated for strings is the same for the table, set, and list representations — they all do the same thing with strings.

In case you're wondering why the amount of memory allocated for strings is slightly larger for the string representation version of the program, it's a consequence of the string comparison needed to check for duplicate line numbers:

```
uses[word][–2 –: *lineno] == lineno
```

The line number is an integer, which must be converted to a string for the comparison. Since a string comparison operation that succeeds returns its right argument as a string, space for this string must be allocated. In any event, it's a minor issue.

The really significant figures are the required sizes of the block regions. 65,024 is the maximum size a block region can be in MS-DOS. Thus, only the string version of the concordance program runs to completion under MS-DOS for the input text we used in this test. (Version 8.5 of Icon supports multiple block and string regions, allowing larger texts to be processed in MS-DOS.)

Of course, memory utilization isn't the only issue. If, for example, the string version of the program were much slower than the other versions, that would be a cause for concern.

We ran the programs on a Sun 4/490 and got these results:

tables	1.250 sec.
sets	1.161 sec.
lists	1.096 sec.
strings	1.083 sec.

There's not all that much difference in the running speed of the four versions of the program, although it is interesting that they get progressively faster. The differences in speed can be partly attributed to the progressively smaller amounts of allocation, but the more significant factor is the number of garbage collections they perform: the table representation does 4 garbage collections, the set representation does 3 garbage collections, and the other two do 2 garbage collections.

Reflections

There are several points worth noting in this case study. One is that it clearly can pay to give careful consideration to data representation. And even if you chose an inappropriate one initially, it may not be that hard to change to a better one — there are very few lines of differences in the programs studied here.

If there was any surprise for us, it was that the string representation was the fastest of the lot. We knew it would use less memory, but we expected to pay for it in speed.

Of the four programs, the table, set, and list representations are all quite similar in what they do. The string representation, on the other hand, despite its similarity in form to the other programs, builds the output as it goes, instead of deferring it until after all input is processed. Of course, the strings that are written have to be constructed eventually in any case.

Of the four programs, we prefer the one that uses sets for the natural and elegant way is handles duplicate line numbers.

But we can't argue with the performance of the string representation.

And That Question

Earlier we raised the issue of writing the output directly to avoid the concatenations in the final phase of the program. Here's how it can be done for the set representation:

```
procedure output()
  local word, i

  uses := sort(uses, 3)

  while word := get(uses) do {

    writes(left(word,width))
    numbers := sort(get(uses))

    while i := get(numbers) do
      if *numbers = 0 then write(i)
      else writes(i, ", ")
    }
end
```

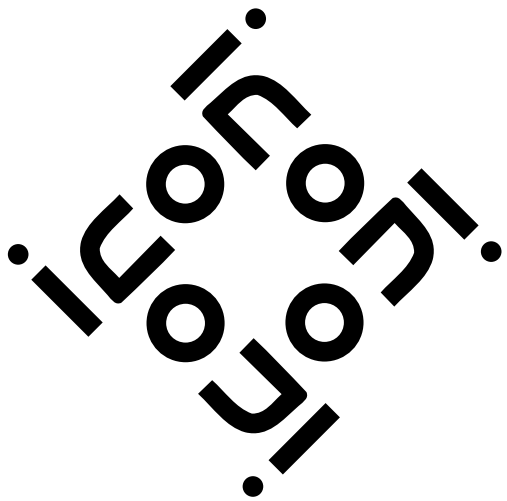
Using this method, the string allocation is reduced from 139,954 bytes to 43,772 bytes, but it doesn't change the other memory use (and it doesn't save a garbage collection for the input text we used for these comparisons).

It also reduces the running time for the set version from 1.161 sec. to 1.075 sec. Since most of the time in this program is spent analyzing the input text and building the structures, it's not surprising that there isn't more improvement in speed.

Note that this method is not applicable to the string version of the program. On the other hand, the figures given above were run on a version of Icon that does not have the optimization for string concatenation that is so effective when a string is appended to the last allocated string [1].

Reference

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



Modeling Icon Functions

There's no substitute for actually implementing a feature of a programming language to gain an understanding of how it works. And, for Icon, you often can do the implementation in Icon itself and not have to go to a low-level language like C. There's an example of this approach in the article on modeling string scanning in Reference 1.

It's even simpler to write procedures that model Icon's built-in functions. Since a procedure that has the same name as function overloads the function (that is, replaces it), you can try out such procedures in real programs.

Modeling Icon's functions by procedures can be instructive in several ways. In modeling a function, you have to think about details that you might never consider otherwise. This can give you a better understanding of the function — and a better one than you're likely to get reading a description of the function in a book. It may also give you a better appreciation of all the things that Icon takes care of for you. And if you write a procedure to replace a function, you can easily trace it or add instrumentation and diagnostics [2].

If you are going to model a function, there are several ways you might approach it. One is to produce the required functionality using compact and idiomatic Icon programming techniques. The model of tab() given in Reference 3 is an example:

```
procedure tab(i)
  suspend .&subject[.&pos : &pos <– i]
end
```

This procedure has several subtle aspects and probably is not what you'd write if you hadn't seen it before.

A less idiomatic but more easily understood version is

```
procedure tab(i)
  local save_pos

  save_pos := &pos

  if &pos := i then {
    suspend .&subject[save_pos : &pos]
    &pos := save_pos
    }

  fail

end
```

Even this more "open" formulation shows the subtleties of Icon. It's necessary to dereference the substring of the subject before suspension. Otherwise, the procedure tab() would return a variable to which an assignment could be made to change the subject, as in

```
tab(i) := "xxx"
```

This is an interesting idea, but it's not what the function tab() does.

Another way to handle this problem is to introduce an intermediate result, as in

```
result := &subject[save_pos:&pos]
suspend result
```

However, the reason for doing this might be lost on a casual reader, and one of the purposes of modeling a function with a procedure is to make things clear.

Another aspect of modeling functions concerns type checking and conversion. In the procedure above, if i is not an integer or convertible to one, a run-time error occurs in the expression

```
&pos := i
```

That is the appropriate thing to happen, but it's not evident in the modeling. It would be better to put an explicit conversion and test in the procedure:

```
i := integer(i) | runerr(101, i)
```

There's another subtlety in the expression

```
&pos := i
```

If i is non-positive, the value assigned to &pos is automatically converted to the positive equivalent with respect to the length of the subject. Again, it might be better to make this explicit in the procedure. With this, the entire procedure is:

```
procedure tab(i)
  local save_pos

  i := integer(i) | runerr(101, i)

  save_pos := &pos

  if &pos := cvpos(i, &subject) then {
    suspend .&subject[save_pos : &pos]
    &pos := save_pos
    }

  fail

end
```

with the support procedure

```
procedure cvpos(i,s)

  if i <= 0 then i +:= *s + 1
  if 1 <= i <= *s + 1 then return i
  else fail

end
```

The function upto(c, s, i, j) is another good candidate for modeling because it raises issues about default arguments that may not have occurred to you. You know if the last three arguments are omitted (or null) that upto(c) applies to the subject in the current scanning environment at the current position through the end of the subject. But what if s is omitted but i isn't?

Here's the answer (we admit we had to refer to the C code in the actual implementation to be sure we got it right — we weren't sure and we didn't trust the description in the Icon language book):

```
procedure upto(c, s, i, j)
  local k

  if /s := &subject then {
    /i := &pos
    }
  else {
    s := string(s) | runerr(103, s)
    /i := 1
    }

  i := integer(i) | runerr(101, i)
  i := cvpos(i, s) | fail

  if not(/j := *s + 1) then {
    j := integer(j) | runerr(101, j)
    j := cvpos(j, s) | fail
    if i > j then i :=: j
    }

  every k := i to j do
    if any(c, s[k]) then suspend k

  fail

end
```

The actual computations of the positions returned by upto() raises another question about modeling functions. When is it permissible to use one function when modeling another? If you're modeling upto() and any() at the same time, you certainly can't model them in terms of each other.

We prefer to use more primitive operations in modeling functions. For example, the computation in upto() can be done as

```
if !c == s[k] then suspend k
```

This is very inefficient, but efficiency presumably is not the issue here.

The question of using functions in modeling raises other issues. What functions can't be modeled without using other functions? What functions can't be modeled by procedures at all? What then is the smallest set of functions that are needed to support the entire function repertoire of Icon?

If you think about this a little, you'll see that most of Icon's functions can be modeled and modeled without using any other function. However, while you can model write() using writes() (although it's a bit tricker than you might think), you can't model writes() as a procedure.

Assuming all of Icon's operators are available for use in modeling (we'll not try to model *them*), you might find it interesting to go through Icon's function repertoire to identify those that are "basic". We'd be interested to see your list.

Before leaving the subject of modeling functions, let's look at a function what has a particularly interesting implementation: map(s1, s2, s3).

The way this function is actually implemented is important, since efficiency is a significant concern. While efficiency in a procedural model of map() is not an issue, such a

model can serve to illuminate the actual implementation.

The character mapping performed by map() is done by building an array of characters based on s2 and s3 and the indexing into it with the characters of s1 to get the required result.

The significant point is that building the mapping array is a comparatively time-consuming process. To avoid building the mapping array unnecessarily, the arguments s2 and s3 and the mapping array are cached. If map() is called with the same values of s2 and s3 as the last time it was called, the cached mapping array is used instead of building a new one.

This technique is particularly effective because of the way map() usually is used, in a loop with the same second and third arguments, as is typified by

```
while line := map(read(), &lcase, &ucase) do …
```

Here's the complete procedure, patterned after the function as written in C:

```
procedure map(s1, s2, s3)
   local i, result
   static last_s2, last_s3, map_array

   initial map_array := list(256)

   s1 := string(s1) | runerr(103, s1)

   s2 := def_str(s2, string(&ucase)) |
      runerr(103, s2)
   s3 := def_str(s3, string(&lcase)) |
      runerr(103, s3)

   if (s2 ~=== last_s2) | (s3 ~=== last_s3) then {

      last_s2 := s2
      last_s3 := s3

      if *s2 ~= *s3 then runerr(208)

      every i := 1 to 256 do
         map_array[i] := char(i – 1)

      every i := 1 to *s2 do
         map_array[ord(s2[i]) + 1] := s3[i]

      }

   result := ""

   every i := 1 to *s1 do
      result ||:= map_array[ord(s1[i]) + 1]

   return result

end
```

Note that map_array is first set up to map every character to itself and then the mappings of the characters of s2 to s3 are written over the identity mapping.

The procedure def_str() defaults the null value to the appropriate string:

```
procedure def_str(s1, s2)

   if /s1 then return s2
   else return string(s1)

end
```

References

1. "Modeling String Scanning", The Icon Analyst 6, pp. 1-2.

2. "From the Wizards", The Icon Analyst 2, p. 12.

3. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, p. 178.

Command-Line Arguments

Almost all programs require some input data. In addition to input files to process, many programs require some specific information, such as options for processing data. Often this information is needed only when the program begins execution.

If a program runs interactively, it can query the user for the information it needs. But even in these days of sophisticated graphical user interfaces, many programs are run "off line". This is particularly true of UNIX, where pipes and a firmly entrenched computing culture frequently result in one program running another program or one program piping data into another program. In such situations, there is no user to query. Programs written to run in such a way must be written to work without interactive input.

For these situations, or where no better user interface is available, the best method of providing information to a program often is on the command line when the program is executed.

Executing Icon Programs

The details of executing an Icon program and providing it command-line arguments vary somewhat, depending on the operating system on which it's run and on whether you're using the Icon interpreter or the Icon compiler.

To avoid complicating the discussion here, we'll use the UNIX environment for examples. It has the virtue of being free of complicating details and, if you're using Icon in a different environment, you'll easily see the differences.

When using the Icon interpreter, the linker produces an icode file, which is executable on most UNIX platforms. The compiler, of course, produces a real executable file:

```
icont prog
```

and

```
iconc prog
```

produce essentially the same results — they produce an executable file named prog from the source file named prog.icn.

In either case, just entering the program name causes the program to execute. All that's needed is

```
prog
```

Anything that appears on the command line after the program name constitutes arguments to the program. For example, in

```
prog data.log
```

data.log is an argument to prog.

It's also possible to go directly into execution when using either the compiler or interpreter by using the –x option:

```
icont prog –x
iconc prog –x
```

Notice that the –x comes after the Icon source program name or names. It is a separator; arguments to icont and iconc come before the source file names, while arguments to the running Icon program come after the –x, as in

```
icont –t prog –x data.log
```

where –t is an option to icont and data.log is an argument to prog.

In subsequent examples, we'll assume an executable file already exists.

Arguments

When a program is executed from the command line, any arguments following the program name are made into a list of strings. This list becomes the value of the argument of the procedure main() (if it has one).

Command-line arguments are separated by white space (blanks and tabs). For example, if prog.icn begins as

```
procedure main(args)
   .
   .
   .
```

and is executed as

```
prog log.dat log.out
```

the value of args is a list containing the two strings "log.dat" and "log.out", as if args had been created by

```
args := ["log.dat", "log.out"]
```

As always, there are numerous questions of syntax. Suppose, for example, you want a blank as part of an argument. That's easy enough; just put quotes around the argument, as in

```
prog "log.dat log.out"
```

which results in one argument, as if args had been created by

```
args := ["log.dat log.out"]
```

Note that the quotes are not part of the argument string itself.

But what if you want a double quotation mark in an argument? In UNIX, you can enclose the argument containing the double quotation mark in single quotation marks or precede the double quote by a backslash. But what about single quotes and backslashes in arguments?

If you really need such characters in arguments (and there are quite a few "special" characters to worry about), then you need to know how your command-line interpreter (shell) works. Things can get pretty messy — we'll leave it to you to explore these issues if you're not already familiar with the details. For most programs, however, arguments to Icon programs do not need to contain "special" characters and the usual separation of arguments by white space on the command line provides all the capabilities you need.

In most Icon programs, command-line arguments serve one of two purposes:

• They provide the names of files that the program uses.

• They provide options that affect how the program works.

Files

While standard input and output are sufficient and convenient for most programs, some programs need named files. This is true, for example, for platforms on which the translated and untranslated modes of reading and writing files produce different results as the result of line-termination conversion. See Reference 1 for a detailed description of the issues involved.

Since standard input and output are always done in the translated mode, a file must be opened by name inside a program that reads or writes it in untranslated mode. The program fileprnt.icn in the Icon program library provides an example. This program reads a file and prints various representations of each character in it. If the input were taken from standard input on, say, an MS-DOS system, carriage-return/linefeed line terminators would be converted to linefeeds during input and the program would produce erroneous output.

The way to handle this problem is to specify the input file name on the command line:

```
fileprnt data.log
```

where fileprnt.icn begins as

```
procedure main(args)
   input := open(args[1], "u") |
      stop("cannot open file")
                .
                .
                .
```

Options

Many programs can be easily configured to perform computations in different ways. For example, a word-listing program might produce unsorted output, sorted output, sorted output in ascending or descending order, only unique words, and so on.

Such options allow one program to serve many needs. In most environments, the best way to specify options is on the command line.

Since a command line can contain various kinds of information, some method is needed to distinguish options from other command-line arguments.

Most operating systems have conventions for how options are specified, but they usually do not enforce these conventions. For example, in UNIX it is conventional (but by no means universal) to indicate an option by an initial dash in an argument. The letter following the dash identifies the option, and there may be additional modifiers following the letter.

For example, in UNIX

```
wordlist –sa –u
```

might indicate that the word list is to be sorted in ascending order with only unique words listed. Here, a ("ascending") is a modifier for s ("sort").

White space between an option and its modifier usually is optional, so that

```
wordlist –s a –u
```

is equivalent to the command line above.

In MS-DOS, an initial slash is conventional, although some programs also accept dashes as well. Thus, in MS-DOS the options might appear as follows:

```
wordlist /sa /u
```

If you're writing Icon programs for your own use, it hardly matters what syntax you use for options, although choosing a consistent style is certainly worthwhile. If you're writing programs for others, you probably will want to use a syntax that is familiar to them.

In the Icon program library, most programs follow a UNIX-like convention and use the procedure options() in the Icon program library to process options. The virtue of using options() is that it provides a consistent interface. It also handles all kinds of details (options can be very complex) as well as error-checking.

options() is called with two arguments — a list of strings (normally the argument to main()) and a string specifying allowable options and the nature of their modifiers, if any:

```
options(args, optstring)
```

Each character in optstring corresponds to an option that the program supports. If an option has an modifier, the type of the modifier is given by the next character:

- : The modifier of the option is a string.
- + The modifier of the option must be an integer.
- . The modifier of the option must be a real number.

For example,

```
options(args, "s:u")
```

indicates that the options s and u are supported options, that the modifier of s must be a string, and that u has no modifier.

options() returns a table containing entries for all the options that appear in args. The corresponding values are the modifiers for the options, if given, or 1 if no modifier is specified. The table's default value is null. For example,

```
opts := options(args, "s:u")
```


The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

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

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

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

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

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA
and

The Bright Forest Company
Tucson Arizona

```
order := \opts["s"]
```

assigns the modifier of the s option to order if s is specified in args.

options() removes options and their modifiers from args but leaves other arguments in args. Thus, for example, if there are both options and file names, the file names remain in args after options() processes the options. Options and other arguments, such as file names, can be freely intermixed and there is no notion of order in the table returned by options().

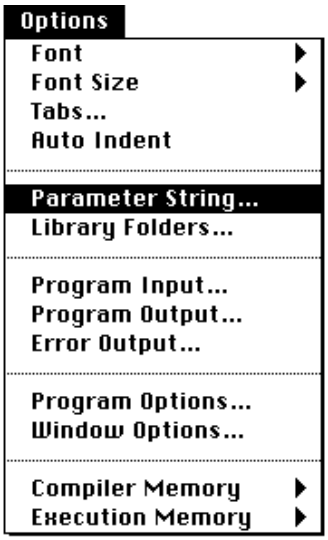
The argument "–" standing alone is not considered to be an option and is left in args. UNIX applications typically use "–" to stand for standard input or standard output.

Gregg Townsend comments: "Although there are a few well-known exceptions, it is also conventional in UNIX for only input files to be named as command arguments, with output redirection handled by special shell syntax. A program that interprets an argument as an output file name may give someone an unpleasant surprise. The ideal UNIX filter program accepts any number of input files, concatenating them as necessary, and reads standard input if no file name is given."

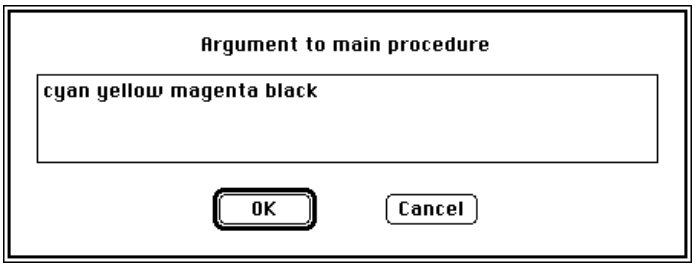
Parameter Strings in ProIcon

Most platforms have a command-line processor from which Icon can be run, even if they also support a graphical user interface. The Macintosh (except under its MPW subsystem) is the notable exception. Most Macintosh applications have no way of specifying options when they are launched, but instead rely on menus and dialogs after they are launched.

ProIcon [2], which runs under the standard Macintosh graphical user interface, offers compatibility with Icon programs designed for use in the command-line mode by means of the Parameter String … entry in its Options menu:

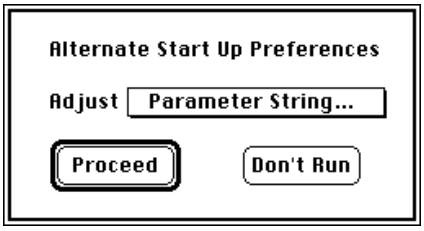


A text-entry box pops up in which arguments can be entered:



The values entered are treated as they would be by a command-line processor. For example, given the parameter string shown in the box above, the argument of main() is a list of the four strings "cyan", "yellow", "magenta", and "black".

You also can set the parameter string when you launch a ProIcon program by holding down the option key. A dialog box comes up:



See the ProIcon manual for more details [2].

Argument Files

Sometime it's useful to have the arguments for a program in a file (one argument per line) rather than entering all the arguments on the command line. This is particularly useful if there are many arguments or if they are syntactically complicated.

The common syntax for programs that support argument files is to use the character @ as a prefix to a file name. For example,

```
prog @argfile
```

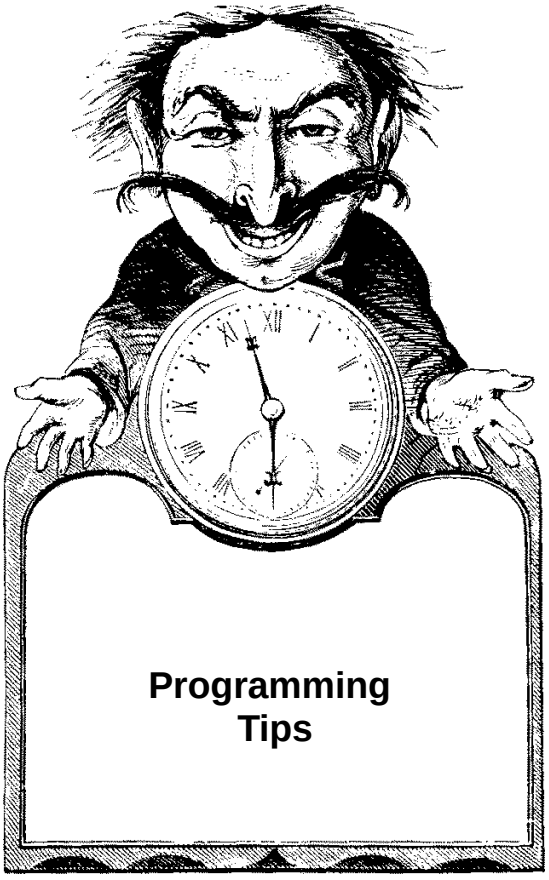
could be used to indicate that the arguments for prog are in argfile.

The latest version of options() in the Icon program library supports such argument files. It allows argument files to be freely interspersed with other arguments and allows argument files to specify other argument files.

References

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 135-136.

2. *The ProIcon Programming Language for the Apple Macintosh Computers; Version 2.0*, The Bright Forest Company, Tucson, Arizona, 1990.



Programming Tips

Exploiting Expression-Based Syntax

Most (but hardly all) imperative programming languages distinguish between expressions and statements. In these languages, an expression produces a value, but a statement just performs some action and has no associated value. Typical statements in such a languages are

```
if expr1 then expr2 else expr3
```

and

```
return expr
```

while typical expressions are

```
expr1 * expr2
```

and

```
expr1 = expr2
```

A statement is syntactically illegal where an expression is expected, which can be annoying at times. For example, you might want to return the value of the expression evaluated in an if-then-else statement.

There are some programming languages, including Icon, in which there are no statements; everything executable is an expression. This means that a construction like

```
if expr1 then expr2 else expr3
```

is an expression and has a value. As you might expect, the value of if-then-else is the value of the selected expression.

In Icon, unlike most other programming languages, an expression does not simply have a value; it has an *outcome*. The word outcome is used here in a technical sense and includes the possibility of no value (failure), a single value, or many values (generation). Thus, in Icon, the outcome of

```
if expr1 then expr2 else expr3
```

is the outcome of *expr2* or *expr3*, whichever is selected.

The outcome of an if-then-else expression usually is not used, but it can be. For example,

```
if expr then write("succeeded") else write("failed")
```

can be recast as

```
write(if expr then "succeeded" else "failed")
```

Consider also

```
if i < j then every write(i to j) else every write(j to i)
```

which writes the integers from i to j or j to i. Since outcome includes generation, this expression can be recast as

```
every write(if i < j then i to j else j to i)
```

We don't necessarily recommend using Icon's expression-based syntax in this way; such expressions tend to be hard to understand. But you should know they are possible and how they work, if only because you may run into them in Icon programs written by others.

There's one place where the "factoring out" that's possible in Icon can be particularly helpful. Consider

```
case expr of {
  1: return expr1
  2: return expr2
  3: return expr3
     :
  }
```

A more compact form is

```
return case expr of {
  1: expr1
  2: expr2
  3: expr3
     :
  }
```

Back Issues

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

From Our Readers

We're always glad to hear from readers of the Analyst. Some of the issues they've raised are listed below.

Who writes the articles for the Analyst? Why are there no bylines? I'd like to know who writes what.

To date, we, the editors, have written all the articles for the Analyst, which is why you don't see bylines. Where there's contributed material, such as programming tips, we always provide attribution.

I'd like to contribute an article to the Analyst. How do I do that? What are the guidelines? How far in advance of publication do you need it?

As indicated in the answer above, we've not published any contributed articles in the Analyst to date. We'd generally prefer to publish such contributions in the *Icon Newsletter*, which has a much larger readership than the Analyst. We do not, however, rule out contributed articles in the Analyst. Send us your contribution and we'll let you know where we think it should appear.

As to the mechanism for contributions, we need the text in machine-readable form — we won't undertake to keyboard contributions from printed text. If there are special typographical requirements, a printed copy with notes, if necessary, should accompany the machine-readable material.

We can handle a wide variety of magnetic media in various formats. We do our work on a Macintosh, but we easily can handle MS-DOS diskettes. Plain ASCII text usually is easiest, although we can handle some word processor formats. For anything other than ASCII text on a Macintosh or MS-DOS diskette, please check with us in advance.

As to publication schedules, the Analyst is written from four to 10 months in advance, depending on the time of year. We try to get ahead during the summer, so that all issues through the following June are done in draft by August.

In the case of the *Newsletter*, we begin assembling the next *Newsletter* as soon as one is published. We do not work farther ahead as we do for the Analyst.

I use SNOBOL4 as well as Icon. I'd like to see articles in the Analyst on SNOBOL4, especially on pattern matching.

The Analyst is about Icon and most of our readers subscribe principally because of an interest in Icon. While we don't plan to include articles in the Analyst on SNOBOL4, *per se*, we have started an article on comparing SNOBOL4 pattern matching to string scanning in Icon. We're not particularly pleased with what we have so far, but we'll continue to work on it.

In the Icon book you show how to construct trees, dags, and graphs using Icon data structures. It think it would be interesting to readers of the Analyst if you'd show how to construct and manipulate some other, more specialized data structures.

Good idea — we'll add that to our queue (actually, it's a heap).

I never have really understood co-expressions. The second edition of the Icon book is better on this subject than the first edition, but I'm still a little lost. How about an article on co-expressions in the Analyst?

We doubt we can do much better by way of describing co-expression in an article in the Analyst than we did in the second edition of the Icon book. Maybe we can find a program that uses co-expressions in an essential way and include it in our "anatomy" series.

It seems to me there's a lot of material in each issue of the Analyst. Doesn't it get difficult to put an issue together every two months? Don't you run out of material? How long do you expect to keep this up?

Very good questions; we ask ourselves such things from time to time. It does take a lot of time to put together an issue of the Analyst. We try to avoid the pressure of publication deadlines by working well ahead (see the answer to a previous question).

We have an essentially endless supply of some features, such as program anatomies. For other features, such as the programming tips, we're finding it increasingly difficult to come up with new, interesting material.

The material that is the easiest and most enjoyable to write about comes from new things going on with Icon, such as the optimizing compiler, X-Icon, and program visualization.

As to how long we can "keep this up" … we don't know. At present, it is a demanding but rewarding occupation from which we have no plans to retire.

What's Coming Up

In the next issue of the Analyst, we'll introduce a new feature — exercises to test your programming skill. The subsequent issue will have our solutions. If this feature works out well, we'll continue it on an irregular basis.

Next time we'll also have another in the series on the anatomies of programs. The next one will be a suffix calculator, which illustrates string invocation.

We'll also continue the series of articles on the optimizing compiler for Icon. The next article will describe how the compiler is organized and give a peek inside at some of the details.