The Icon Analyst

In-Depth Coverage of the Icon Programming Language

October 1991
Number 8

In this issue ...

String Synthesis … 1
An Imaginary Icon Computer … 2
Augmented Assignment Operations … 7
The Icon Compiler … 8
Programming Tips … 12
What's Coming Up … 12

String Synthesis

In the last issue of the Analyst, we posed the problem of implementing a string-synthesis facility for Icon, using the ideas given earlier about modeling the string-scanning control structure.

Our solution is given below. First we need procedures analogous to the procedure used for modeling string scanning. In addition to a subject, there's now an "object", which is the result of string synthesis. There now also are two positions, one in the subject and one in the object. The global identifiers subject, object, s_pos, and o_pos are used for these four "state variables" in the procedures that follow. (In a real implementation, these would be keywords.)

The string synthesis control structure is modeled as

expr1 ? *expr2* → Eform(Bform(*expr1*),*expr2*)

The procedures Bform() and Eform() are very similar to Bscan() and Escan() used in the last issue of the Analyst for modeling string scanning. There just are two additional state variables to maintain. The subject gets its value from *expr1* as before, while the object starts out as the empty string. Both positions start at 1.

```
record XformEnvir(subject, s_pos, object, o_pos)

global subject, object, s_pos, o_pos

procedure Bform(e1)
   local OuterEnvir

   OuterEnvir :=
      XformEnvir(subject, s_pos, object, o_pos)
   subject := e1
   object := ""
   s_pos := o_pos := 1

   suspend OuterEnvir
   subject := OuterEnvir.subject
   object := OuterEnvir.object
   s_pos := OuterEnvir.s_pos
   o_pos := OuterEnvir.o_pos
   fail

end

procedure Eform(OuterEnvir, e2)
   local InnerEnvir

   InnerEnvir :=
      XformEnvir(subject, s_pos, object, o_pos)
   subject := OuterEnvir.subject
   object := OuterEnvir.object
   s_pos := OuterEnvir.s_pos
   o_pos := OuterEnvir.o_pos

   suspend InnerEnvir.object

   OuterEnvir.subject := subject
   OuterEnvir.object := object
   OuterEnvir.s_pos := s_pos
   OuterEnvir.o_pos := o_pos
   subject := InnerEnvir.subject
   object := InnerEnvir.object
   s_pos := InnerEnvir.s_pos
   o_pos := InnerEnvir.o_pos
   fail

end
```

Most of the procedures specified in the last issue of the Analyst are straightforward. Care must be taken, however, to assure that values assigned to the positions are in range — this is done automatically for the keyword &pos, but it must be done explicitly for s_pos and o_pos. The procedures smove() and omove() illustrate this:

```
procedure smove(i)

   if s_pos + i >= 1 then
      suspend .subject[.s_pos:s_pos <– s_pos + i]

end

procedure omove(i)

   if o_pos + i < 1 then fail
   suspend .object[.o_pos:o_pos <– o_pos + i]

end
```

Note that smove() and omove() are the same except for the use of different variables.

In addition, non-positive specifications for positions must be converted to positive ones. The procedure cvpos() takes care of this:

```
procedure cvpos(i, s)
   if i <= 0 then i +:= *s + 1
   if 1 <= i <= *s + 1 then return i
   else fail
end
```

This conversion is needed in all procedures whose arguments are position specifications:

```
procedure spos_(i)
   return cvpos(i, subject) = s_pos
end
procedure stab(i)
   suspend .subject[.s_pos:s_pos <–
      cvpos(i, subject)]
end
procedure opos_(i)
   return cvpos(i, object) = o_pos
end
procedure otab(i)
   suspend .object[.o_pos:o_pos <–
      cvpos(i, object)]
end
```

Again note the similarity of the procedures that deal with the subject and the object.

The three interesting procedures related to synthesis are:

```
procedure xswap()
   suspend (object <–> subject) & (o_pos <– 1) &
      (s_pos <– 1) & &null
end
procedure odelete(i)
   suspend (object[o_pos+:i] <– "") & .opos
end
procedure oplace(s)
   suspend (object[o_pos:o_pos] <– s) &
      .(o_pos <– o_pos + *s)
end
```

In all three, conjunction and reversible assignment are used to assure that the state variables are restored to their former values if a suspended procedure call is resumed.

What Next?

If you're interested in programming language design, you may wish to try your hand at adding additional string synthesis procedures.

Another possibility is a facility that combines analysis and synthesis so that the subject is transformed as it's analyzed. While this kind of string transformation has conceptual appeal, it presents the problem of keeping track of where you are in a subject whose length may be changing. If you come up with some ideas, you can try them out without a great investment in time and effort by using the modeling techniques we've shown.

An Imaginary Icon Computer

As mentioned in the last issue of the Analyst, the interpretive implementation of Icon is based on the concept of an imaginary Icon computer that performs the operations needed to execute an Icon program.

The instruction set of this imaginary computer is a bit strange — it's not something you'd want to build in hardware. Rather, this imaginary computer forms a conceptual bridge between the semantics of Icon and the architecture of conventional computers on which Icon must run. This implementation technique is an old one. The first language in which the method was well documented was BCPL [1]. The macro implementation of SNOBOL4 also used this technique [2].

Virtual Machine Language

The code for this Icon computer is called virtual machine code and is described in some detail in the Icon implementation book [3]. We won't attempt to describe virtual machine code in detail here, but a few of its characteristics are worth note.

First, the imaginary Icon computer is stack-based. That is, the operands of an operation are pushed on to a stack. The operation pops its operands off the stack and pushes its result in their place.

One curious aspect of the virtual machine language is that there is a separate instruction for every Icon operator — not just ones for arithmetic operations. There is an instruction for subscripting, an instruction for activating a co-expression, and so on.

On the other hand, there is just one instruction for invoking all functions and procedures. The difference between the handling of operators and functions is a reflection of the fact that Icon operators are fixed and their meanings cannot change during program execution:

```
i + j
```

always means addition. On the other hand, functions and

procedures are first-class values. They start out as the values of global identifiers, but other values can be assigned to these identifiers. Consequently,

```
write(s)
```

may be the invocation of the built-in function for writing, or it may be something entirely different if another value has been assigned to write.

The virtual machine language does not attempt to cope with generators. There is a single virtual machine instruction for the element generator (!x) just as there is one for the size operation (*x). It's left to the instruction (that is, its execution by the imaginary Icon computer) to handle what goes on in generation.

As with any real computer, there are lots of other instructions: instructions to push values on the stack, instructions to reference variables, and instructions to transfer control between places in a virtual machine program.

You can look at the virtual machine code if you wish — it's found in ucode files. As described in the article in the preceding Analyst, ucode is the format used in procedure libraries. If you have a library of ucode files (perhaps from the Icon program library), you can print them or examine them with a text editor — they are plain text files. You also can create ucode files using the –c option to the Icon interpreter, as in

```
icont –c sum.icn
```

This produces a pair of ucode files, sum.u1 and sum.u2.

The .u1 file contains virtual machine code for the program, while the .u2 file contains global program information.

An Example

Consider the following Icon program that produces the sum of real numbers given in the standard input file. It also removes any leading or trailing white space and ignores lines that do not contain valid real numbers:

```
procedure main()
   total := 0.0
   every total +:= check(!&input)
   write("Total = ", total)
end

procedure check(s)
   s ? {
      tab(many(' \t'))
      return real(trim(tab(0), ' \t'))
      }
end
```

If this program is in the file sum.icn and is translated with the –c option, the file sum.u1 that results is as shown in the box at the right.

```
proc main                                          1
        local   0,000000,total
        local   1,000000,check
        local   2,000000,write
        con     0,004000,0.0
        con     1,010000,8,124,157,164,141,154,040,075,040
        declend                                    2
        filen   sum.icn
        line    1                                  3
        mark    L1
        pnull
        var     0
        real    0
        line    2
        asgn
        unmark
lab L1
        mark    L2
        mark0
        pnull
        var     0
        dup
        var     1
        pnull
        line    3
        keywd   18
        bang
        invoke  1
        plus
        asgn
        pop
lab L3
        efail
lab L4
        unmark
lab L2
        mark    L5
        var     2
        str     1
        var     0
        line    4
        invoke  2
        unmark
lab L5
        pnull
        line    5
        pfail                                      4
        end

                                                   1
proc check
        local   0,001000,s
        local   1,000000,tab
        local   2,000000,many
        local   3,000000,real
        local   4,000000,trim
        con     0,020000,2,040,011
        con     1,002000,1,0
        declend
        line    7
        mark    L1
        var     0
        line    8
        bscan
        mark    L2
        var     1
        var     2
        cset    0
        line    9
                .
                .
                .
```

sum.u1

There is a section of virtual machine code for each procedure (main() and check() in this program).

❶ Each procedure begins with proc, followed by information about identifiers and constants used in the procedure.

Identifiers are indicated by local, followed by three comma-separated fields. The first field associates an integer (starting at 0) with the identifier. The second field contains information about the identifier, such as if it is explicitly declared to be local (most digits in this field are unused, left over from a time when it was not known what information would be needed). The third field gives the identifier name. Thus, total is identifier 0 in the main procedure.

Next the constants used in the procedure are given. The real number 0.0 is constant 0, while the string "Total = " is constant 1. ASCII codes (or EBCDIC codes on IBM mainframes) are used for string constants, since they (unlike identifiers) can contain characters like linefeeds that would confuse the text of the ucode file if given literally.

The virtual machine instruction declend indicates the end of the heading information for the procedure.

❷ The instruction filen gives the name of the source-language file from which the ucode was compiled. The argument of the line instruction is the source-program line number. File and line information is used in run-time error messages and by the keywords &file and &line. The virtual-machine instructions for the executable code in the procedure follow.

❸ The mark instruction indicates the beginning of a bounded expression. Its argument, L1, is the label of the code to which to branch in case the expression that follows fails. The expression here, a simple assignment, cannot fail, but the translator is not smart enough to account for this.

The instruction pnull pushes a null descriptor to provide a place for the result of the upcoming operation. The var instruction pushes a variable descriptor corresponding to the number given in its argument, and real pushes a real descriptor corresponding to its argument. The asgn instruction performs the assignment using the descriptors on the stack. The argument descriptors are removed, and the result of the assignment (the variable) replaces the null descriptor that was pushed earlier. The result is execution of the expression

```
total := 0.0
```

The unmark instruction undoes the effects of the mark instruction. Virtual-machine instructions for the remaining expressions follow. See Reference 3 if you're interested in more details of the virtual-machine instructions and what they do.

❹ The code for the procedure is terminated by the end instruction. Note that pfail is the last instruction in the body of the procedure, ensuring that if control flows off the end of the procedure body, a call of the procedure will fail, as it does in the procedures in the example here.

Most of the content of a program — its procedure declarations and executable code— is in the .u1 file. The .u2 file, which contains global information about the program, usually is short, as shown here:

```
version         U8.3.000
impl            local
global          2
        0,000005,main,0
        1,000005,check,1
```

sum.u2

The version identification is provided so that the linker can check that the ucode is compatible with its data.

The line

```
impl            local
```

stands for "implicit local" and means that an undeclared identifier will be local unless the linker finds a global declaration for it in another ucode file. If the program has been compiled with the –u option, the corresponding line in the .u2 file would have been

```
impl            error
```

instructing the linker to issue warning messages for undeclared identifiers for which there are no global declarations (they are still made local).

The remaining lines in the .u2 file give the global identifiers with encoding information. Here the only global identifiers are those for the two procedures in the program.

The Linker

Although the linker is not really part of the imaginary Icon computer, it's an essential part of the implementation. The linker processes ucode files, perhaps from several separate translations, resolves the scope of undeclared identifiers, and produces a binary icode file containing the virtual-machine code and data in a form that can be processed by an interpreter.

An icode file is a memory image; it is read into memory when the Icon interpreter is run and "executed".

Since an icode file is in a binary format, it can't be printed or viewed in a text editor to see easily what it contains. The Icon linker, however, can be configured with a debugging option in which a text file showing the icode can be obtained.

It's necessary to have the source code for the linker and compile it with the debugging option. If you have the source code for Icon and want to try this, add

```
#define DeBug
```

to src/h/define.h and recompile icont. With the linker that results, the command line option –L to icont causes a file with the suffix .ux to be produced. For example

```
icont –L sum.icn
```

produces a file sum.ux showing the contents of the icode file.The result is shown on the next page. *Note:* The format of

❶

```
0:   3                                                    ❷

     000 000 000 000 000 000 000 000
               ( 0.0 )

16:  6                                                    ❸
     44
     Z+60
     0
     1
     0
     0
     4          S+0              # main                   ❹
     5          S+20             # total

60:  67         L1               # mark                   ❺
68:  69                          # pnull
72:  83         0                # local
80:  75         *–88             # real                   ❻
88:  1                           # asgn
92:  78                          # unmark
L1:
96:  67         L2               # mark
104: 85                          # mark0
108: 69                          # pnull
112: 83         0                # local
120: 52                          # dup
124: 84         1                # global
132: 69                          # pnull
136: 62         20               # keywd
144: 2                           # bang
148: 61         1                # invoke
156: 30                          # plus
160: 1                           # asgn
164: 70                          # pop
L3:
168: 53                          # efail
L4:
172: 78                          # unmark
L2:
176: 67         L5               # mark
184: 84         2                # global
192: 77         8,S+32           # str
204: 83         0                # local
212: 61         2                # invoke
220: 78                          # unmark
L5:
224: 69                          # pnull
228: 68                          # pfail

232: 4
     2
     000 000 002 000 000 000 000 001

272: 6
     44
     Z+316
     1
     0
     0
     0
     5          S+14             # check
     1          S+49             # s

316: 67         L1               # mark
324: 81         0                # arg

332: 44                          # bscan
336: 67         L2               # mark
344: 84         3                # global
352: 84         4                # global
360: 51         *–136            # cset
                .
                .
                .
484: 0                           # record blocks          ❼
488:                             # record/field table

488: 22000000006   Z+16          # main                   ❽
496: 22000000006   Z+272         # check                  ❾
504: 22000000006   –72           # write
512: 22000000006   –65           # tab
520: 22000000006   –35           # many
528: 22000000006   –52           # real
536: 22000000006   –67           # trim

544: 4          S+0              # main                   ❿
552: 5          S+14             # check
560: 5          S+26             # write
568: 3          S+51             # tab
576: 4          S+55             # many
584: 4          S+60             # real
592: 4          S+65             # trim

600: 060        S+041                                     ⓫

608: 060        001                                       ⓬
616: 088        002
624: 136        003
632: 212        004
640: 228        005
648: 316        007
656: 332        008
664: 368        009
672: 428        010
680: 468        008
688: 480        012

                                                          ⓭
696: 155 141 151 156 000 125 070 056
704: 062 056 060 060 060 000 143 150
712: 145 143 153 000 164 157 164 141
720: 154 000 167 162 151 164 145 000
728: 124 157 164 141 154 040 075 040
736: 000 163 165 155 056 151 143 156
744: 000 163 000 164 141 142 000 155
752: 141 156 171 000 162 145 141 154
760: 000 164 162 151 155 000 040 011
768: 000

size:       769                                           ⓮
trace:      0
records:    484
ftab:       488
fnames:     488
globals:    488
gnames:     544
statics:    600
strcons:    696
filenms:    600
linenums:   608
config:     I8.3.000
```

sum.ux

.ux files varies somewhat, depending on the version of Icon you're using; the file you get may not be identical to the one here. We've also adjusted white space to make it easier to read.)

The general structure of an icode file, as shown in the text-file representation, has several components:

❶ Relative memory addresses are shown at the beginning of most lines.

As in ucode, icode is organized on a per-procedure basis. Each procedure starts with data local to the procedure: blocks for any real numbers and csets used in the procedure, followed by a procedure block.

❷ In the first procedure, there is a block for the real number 0.0 starting at relative address 0.

❸ The procedure block starts at relative address 16. See Reference 3 for information about the contents of such blocks. The symbol Z ("zero") is used to emphasize that an address is relative to the beginning of the icode.

❹ The symbol S ("strings") indicates an address relative to a block that contains the strings (names, string literals, and so forth) contained in the program. Thus, S+20 is where the string for the name of the identifier total is located. In this example, S is 696. We'll explain how this is determined later.

❺ The virtual machine instructions for a procedure follow its data block. Compare these to the corresponding ucode. In icode files, the numerical codes used to encode virtual-machine instructions are given after the addresses. Arguments, if any, are in the next column, and the names of the instructions are given in comments. For example, the numerical code for mark is 67.

❻ The symbol * refers to the current location in the icode (actually, the address of the next instruction, because of when the location is incremented). Thus, *–88 comes out to 0, which is the address of the real block for 0.0.

❼ Data that is global to the program follows the code for the last procedure. Record information comes first (this program has no records).

❽ Descriptors for global identifiers are next (address 488 in this example). The only global identifiers in this program are the ones for the procedures and functions referenced in the program. The first two, for the procedures main() and check(), point to their respective procedure blocks (Z+16 and Z+272)

❾ The remainder of the global descriptors are for the functions used in the program. They have negative identifying values. When an icode file is loaded prior to execution, these negative values are replaced by pointers to their respective function blocks in the run-time system. See Reference 3 for details.

❿ Qualifiers for the names of the global identifiers are next (address 544 in this example). Note that they point to strings in the identifier region.

⓫ A table of icode locations and corresponding file names comes next (address 600 in this example). Since there is only one file for this program, there is only one entry.

⓬ Next there's a table of icode locations and corresponding program line numbers (address 608 in this example). File names and line numbers are used in diagnostic messages and by &file and &line.

⓭ The strings for the program come next (address 696 in this example), with each character shown by its octal value. Strings are pooled. That is, there is only one instance of a string, even if it appears several times in the program. Note that strings are null-terminated as is conventional for C, the language in which the Icon interpreter is written.

⓮ The last part of the .ux file gives information about its size (769 bytes in this example) and the relative addresses of global data in it. At the end is the icode version number. In the actual icode file, this block of information appears first. Space is reserved for it and the icode file is repositioned to write this header after all the other information has been written. We'll explain the reason for this in a subsequent article.

What Does This All Mean?

We've covered a lot of detail here. And there's a lot more you'll need to study if you really want to understand the imaginary Icon computer.

Our purpose in providing all this information is to give you an idea of the conceptual framework for the interpretive implementation of Icon: How the implementation of Icon can be accomplished in terms of an albeit unusual and nonexistent "Icon machine", how this view is used in the actual implementation, and how the implementation on a real computer can be accomplished.

This is just one way to implement a programming language with features that are far from those found in real computer hardware. But it has proved effective for several programming languages and is worth knowing if you are interested in implementation techniques.

In a future article, we'll go one step further and describe what the Icon interpreter does with an icode file and explain a little about the interpreter itself.

References

1. "The Portability of the BCPL Compiler", Martin Richards, *Software — Practice & Experience*, Vol. 1, No. 2, 1971, pp. 135-146.

2. *The Macro Implementation of SNOBOL4; A Case Study of Machine-Independent Software Development*, Ralph E. Griswold, W. H. Freeman, San Francisco, California, 1972.

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

Augmented Assignment Operations

Icon provides "augmented" assignment operations for all its binary operations (except the assignment ones). Superficially, an augmented assignment operation is just an abbreviation for the common case when the value assigned to a variable is the result of an operation on that variable. The commonest example of this situation is incrementing a counter, as in

```
count := count + 1
```

which can be written with augmented assignment as

```
count +:= 1
```

Many programming languages support some form of augmented assignment, but most support it for only a few operations such as addition and subtraction.

To use augmented assignment correctly and effectively, it's important to realize just what augmented assignment does. For a binary operation ⊗,

expr1 ⊗:= *expr2*

is equivalent to

expr1 := *expr1* ⊗ *expr2*

except that *expr1* is evaluated only once.

There are two important points here: (1) *expr1* can be any expression that produces a variable, and (2) the augmented form is equivalent to the non-augmented form with *expr1* as its first (left) operand.

The fact that *expr1* is the left operand does not matter for commutative operations such as addition and intersection, but it's important for non-commutative operations such as division and concatenation. Thus, you can use augmented assignment with concatenation to append but not to prepend to a string-valued variable.

The fact that *expr1* is evaluated only once in augmented assignment is not particularly significant if *expr1* is just an identifier, but it can be very significant if *expr1* is a more complicated expression.

One issue here is efficiency. While the amount of time it takes to "evaluate" an identifier is negligible, the amount of time required for table look up can be significant. Thus, there is a potential saving in execution time for expressions like

```
wordcnt[word] +:= 1
```

in place of

```
wordcnt[word] := wordcnt[word] + 1
```

The other important aspect of evaluating *expr1* only once occurs for expressions that may have different values when evaluated at different times. Of course, you'd not write something like

```
wordcnt[read()] := wordcnt[read()] + 1
```

to increment the count of a string read in from a file — the two instances of read() clearly read two lines of input. If you don't use augmented assignment, you need a temporary result, as in

```
line := read()
wordcnt[line] := wordcnt[line] +1
```

but of course, augmented assignment makes that unnecessary:

```
wordcnt[read()] +:= 1
```

There are more subtle situations in which an expression may produce different results at different times. For example, suppose tally is a list of integers. Then

```
?tally +:= 1
```

increments a randomly selected element of tally, but

```
?tally := ?tally + 1
```

does something quite different: It usually sets one element of tally to one plus *another* element of tally. It may even *decrease* the value of an element of tally.

It's worth remembering that the left operand of assignment can be a generator. For example,

```
every (i | j | k) := 0
```

sets i, j, and k to 0. This kind of construction can be used to add the same value to several variables, as in

```
every (i | j | k) +:= 1
```

This idiom is even more useful in the case of structures. For example,

```
every !tally +:= i
```

adds i to every element of tally. Similarly,

```
every wordcnt[key(wordcnt)] +:= 1
```

increments every count in wordcnt.

It's natural to use augmented assignment for operations like addition, subtraction, and string concatenation. There even are cases where it's worth using augmented assignment for multiplication and division. For other operations, the use of augmented assignment may not be so obvious. For example,

```
max <:= count
```

sets max to count if max is less than count. It's equivalent to

```
max := max < count
```

noting that the comparison

```
max < count
```

produces the value of count if max is less than count, but fails otherwise (so that max is not changed).

This formulation *is* a bit convoluted; the more conventional

```
if max < count then max := count
```

is easier to understand for most persons.

Another potential use of augmented assignment that often is overlooked occurs in string scanning:

```
s ?:= expr
```

evaluates *expr* in the context of the subject s and assigns the result to s.

In using augmented string scanning, it's important to remember that the value produced by *expr* is what's assigned to s. For example,

```
text ?:= {
   move(5)
   tab(0)
  }
```

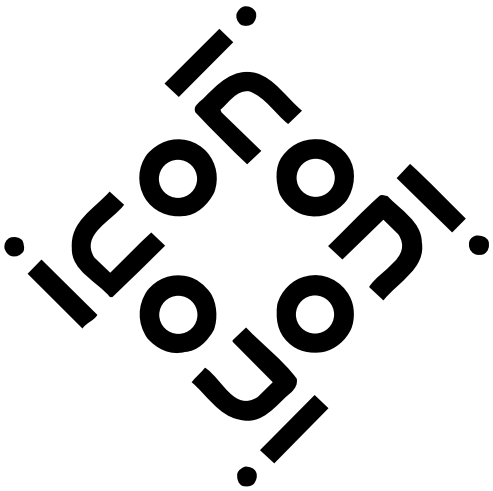
removes the first five characters of text, provided text is that long. (It doesn't change text if text is shorter than that.) Similarly,

```
text ?:= move(5)
```

truncates text to no more than five characters.

This operation is so simple that you might wonder if it wouldn't be better to use

```
text := text[1+:5]
```



Well, perhaps, although string scanning is faster than subscripting in this case (surprise?). And, once you're used to string scanning, it usually "feels better" than lower-level operations such as subscripting. If you know SNOBOL4, you might reflect on this — SNOBOL4 (in its standard form) doesn't have low-level string-analysis functions; everything must be done using pattern matching.

Augmented string scanning offers a special advantage when applying the same scanning expressing to several string-valued variables, as in

```
every (text1 | text2 | text3) ?:= move(5)
```

It's even more interesting if you have, for example, a list of strings, all of which are to be processed in the same way, as in

```
every !wordlist ?:= move(5)
```

Compare this to the alternative formulation that doesn't use augmented string scanning:

```
every i := 1 to *wordlist do
  wordlist[i] := (wordlist[i] ? move(5))
```

Note that generation can't be used here because the same expression must be evaluated twice if augmented string scanning is not used — an expression such as

```
every !wordlist := (!wordlist ? move(5))
```

does not do what you want at all, since suspended generators are resumed in a last-in, first-out fashion. Here the second instance of !wordlist generates all its results for every result generated by the first instance of !wordlist. You might want to figure out what happens.

The Icon Compiler

This is the first of a series on articles on the new optimizing compiler for Icon [1]. We'll start by discussing general issues. In subsequent articles, we'll have more to say about how the compiler works, how to use the compiler, what its advantages and limitations are, and what may be coming in the future.

Language Features and Efficiency

Sophisticated programming language features, especially novel ones, usually are difficult to implement efficiently. They often lead to poor execution performance, especially in early implementations.

History shows, however, that implementation techniques often can be found to overcome these problems.

The implementations of SNOBOL4 provide some of the best examples of this. The first implementation of SNOBOL4, which is still in use, is interpretive [2]. Several

other increasingly efficient implementations followed. The SPITBOL implementation [3], which combines compilation with many clever devices, runs almost 10 times faster than the original interpreter.

Icon abounds with sophisticated features. The main features of Icon that present implementation problems and that impact execution speed are:

1. Type information is not declared in Icon programs, but there is a strong run-time type system. Values have types but variables do not. Any variable can take on a value of any type at run time and the type can change as program execution continues. Structures can be heterogeneous, containing values of different types. However, operations check the correctness of the types of their arguments, performing automatic conversion if possible or issuing error messages if a type cannot be converted to the expected one. Some operations are polymorphous, performing different types of computations depending on the types of their arguments.

2. Expression evaluation is more complex in Icon than in most programming languages. An expression can generate a sequence of values or no value at all (failure). An expression that is capable of producing another value retains its state information by suspending. Goal-directed evaluation causes suspended expressions to be resumed to produce other values. Control backtracking is inherent in this evaluation process. Expression evaluation also involves novel control structures for managing goal-directed evaluation and the production of sequences of values.

3. Functions and procedures are first-class data objects. The functions and procedures that may be invoked during program execution cannot be determined, in general, from the program alone.

4. Strings of characters (not arrays of characters) are central to computation in Icon. Strings are atomic and the operations on them are applicative. Strings are created at run time and may be arbitrarily long. There is a large repertoire of low-level string operations in addition to string scanning.

5. Icon has several different kinds of data structures, with sophisticated organizations and access mechanisms. Lists can be indexed by position, and also as stacks and queues. Sets and tables provide associative look up for values of any type. Structures are created at run time and can be arbitrarily large. They also can change in size during program execution. Structures have pointer semantics; a structure value references the aggregate of values it contains.

6. Storage management is automatic. Data objects are created at run time and space for them is provided automatically. Such data objects vary widely in their characteristics and sizes. Garbage collection is performed automatically to free space occupied by objects that are no longer needed, allowing that space to be re-used for newly created objects.

One fundamental question is whether these features are inherently computationally expensive or whether ways be found to implement them efficiently.

The term efficiency is elusive in this context. If sophisticated features are used in ways that take full advantage of their capabilities, computation may be faster than if conventional techniques are used — simply because the operations are performed internally and don't have to be coded at the source-language level. Consider, for example, writing every position at which one string occurs as a substring of another. Using generators and goal-directed evaluation, all that's needed is

```
every write(find(s1,s2))
```

whereas using conventional control structures, something like this is needed:

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

```
i := 1
while i <= *s2 – *s1 + 1 do {
   if match(s1, s2, i) then write(i)
   i +:= 1
   }
```

In most programming languages, the formulation would be even more complicated, but the point should be clear.

But what if all of the capabilities of a feature are not used? Is there an execution penalty for the unused capabilities anyway? For example, is the execution of

```
i := 1
```

in Icon penalized for generators and goal-directed evaluation, even though they are not used?

It seems clear, at least, that a clever implementation of expression evaluation should be able to detect situations in which generation and goal-directed evaluation are not needed and produce faster code than for situations in which they are needed. A similar argument applies to many of the other features of Icon that impact its performance. In other words, what optimizations are possible?

These kinds of considerations are the basis for the new optimizing compiler that has been written for Icon. Before going on to discuss the compiler, it's important to understand what it attempts to do and how it relates to the existing Icon interpreter.

Main Implementation Issues

There are three general areas of importance in the implementation of Icon:

- data representation
- generated code
- run-time routines

Data representation is pervasive. The way values are represented in Icon is influenced by its type system, the nature of Icon values, and automatic storage management. Generally speaking, choices of data representation are made first and other aspects of the implementation adapt to these choices.

The distinction between generated code and run-time routines occurs because many operations in Icon are too complex for in-line code and instead are implemented by calls to subroutines.

The data representation used in the Icon interpreter has not changed much since Version 3 of Icon and is described in detail in Reference 4.

The Icon interpreter generates code for an imaginary computer, as described earlier in this issue of the Analyst. It hasn't changed much either since Version 3 of Icon.

The run-time routines for the interpreter have been the subject of much work and they have changed substantially over time, the most recent major change being dynamic hashing used for set and table look up [5].

What the Compiler Does

You can approach the design of a new implementation from the ground up, discarding existing work and doing everything over. In the case of a compiler for Icon, this is impractical, at least in a non-commercial environment — it's too big a project and there are too many complex and interacting issues. In addition, many aspects of the Icon interpreter are well done and it's not obvious how to improve on them in major ways — not that it's impossible, just that better methods are not obvious.

The Icon compiler takes the data representation and run-time routines of the interpreter largely intact and concentrates mainly on the generated code. This means that the optimizing compiler does not have much affect on the portion of the time a program spends in run-time routines. For example, if a program spends most of its time in table lookup or garbage collection, the Icon compiler cannot hope to improve its running speed much.

On the other hand, it's somewhat of an over-simplification to say that the compiler just deals with the generated code. That makes it sound like it's just a question of what kind of code the compiler generates. There's more to it than that. For example, the compiler generates in-line code in some situations in which the interpreter calls run-time routines. The compiler's interface to run-time routines also is somewhat different from the interpreter's, as is the method for allocating space for temporary results in the two systems.

Another way to view the domain of the compiler is through the language features whose implementation it affects. These are mainly the first three listed on the preceding page: how type information is handled, the code for expression evaluation, and the treatment of functions and procedures.

Generated Code

In the first place, the Icon compiler generates C code, not code for an imaginary computer or for any specific real computer. While an imaginary computer has many conceptual advantages, it's not a model that lends itself to optimizations and, being fairly far from the code for any real computer, there's an inherent inefficiency when it's converted to run on a real computer.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)

There are several reasons for producing C code instead of, for example, producing code that is native to a particular computer. Portability is one reason; C compilers exist for almost all real computers that have the capacity to run Icon. C code also has the virtue of being close enough to most real computers that programs written in C usually are quite efficient. By generating code that is then processed by a C compiler, the Icon compiler can take advantage of the additional code optimizations performed by C compilers.

Since the Icon compiler produces C code, two compilations are involved in going from an Icon source-language program to an executable file. The complete process is shown in the diagram at the bottom of this page.

Although the process of compiling an Icon program involves several steps, this is hidden from the user. All that is needed to compile the Icon program prog.icn is

```
iconc prog
```

More to Come

There are two major issues in code generation for Icon: (1) how to produce code for expressions and (2) optimizations.

The first issue may not be obvious. Models for producing code for expressions in conventional languages like Pascal and C are well known and compiler-writing techniques for such programming languages now are almost routine. For example, anyone who has taken a good course in compiler writing should be able to write a compiler for a language like Pascal or C, although it's admittedly a major project, and producing efficient code may require more knowledge and experience than are provided by a typical course in the subject.

On the other hand, until recently there hasn't been a model for producing code for generators, goal-directed evaluation, and related control structures [6]. Incidentally, the existence of an interpreter for Icon isn't much help in this regard — what's relatively easy to do with an interpreter may be baffling in the context of compilation.

Optimization is the soul of compilation. There's no end to what can be done, even for a conventional programming language. A language like Icon provides a veritable garden of opportunity for optimization.

In subsequent articles, we'll explore these topics and attempt to convey the basic concepts without an overwhelming amount of technical detail.

References

1. *The Implementation of an Optimizing Compiler for Icon*, Kenneth Walker, Technical Report TR 91-16, Department of Computer Science, The University of Arizona, 1991.

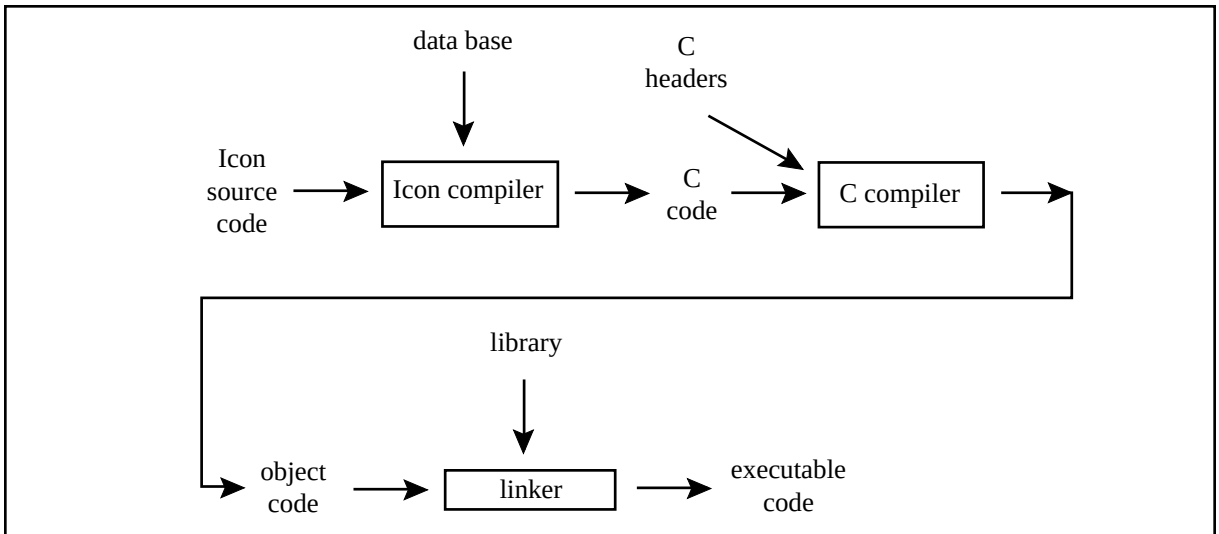
2. *The Macro Implementation of SNOBOL4; A Case Study of Machine-Independent Software Development*, Ralph E. Griswold, W. H. Freeman, San Francisco, California, 1972.

3. "MACRO SPITBOL — A SNOBOL4 Compiler", Robert B. K. Dewar and Anthony P. McCann, *Software — Practice & Experience*, Vol. 7 (1977), pp. 95-113.

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

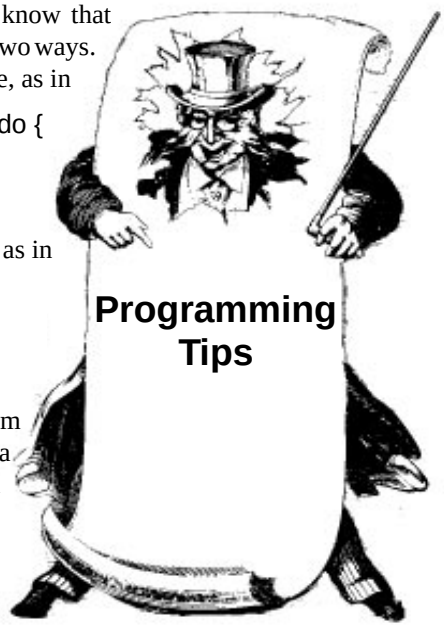
5. *Supplementary Information for the Implementation of Version 8 of Icon*, Ralph E. Griswold, Icon Project Document IPD112, Department of Computer Science, The University of Arizona, 1990.

6. *The Implementation of Generators and Goal-Directed Evaluation in Icon*, Janalee O'Bagy, Technical Report TR 88-31, Department of Computer Science, The University of Arizona, 1988.



Compiling an Icon Program

Element Generation



You probably know that you can read a file in two ways. The conventional one, as in

```
while s := read(f) do {
        .
        .
   }
```

or using a generator, as in

```
every s := !f do {
        .
        .
   }
```

In most cases, the form you chose is largely a matter of personal preference.

There is, however, one aspect of using the generative form that may prove useful in some situations. Since generation can be used on values of several types, you can use the generative form to "read" from structures as well as from files. For example,

```
every s := !input do {
        .
        .
    }
```

works equally well if input is a file or a list.

This technique can be used, for example, to provide a multi-line macro definition facility for input streams. For example, suppose macro definitions have the form

```
#begdef boilerplate
        .
        .
#enddef
```

where the lines between #begdef and #enddef consist of text associated with the macro boilerplate and suppose

```
>boilerplate
```

in the input stream indicates an invocation of the macro boilerplate, indicating that the text associated with name is to be inserted in the input stream at this point.

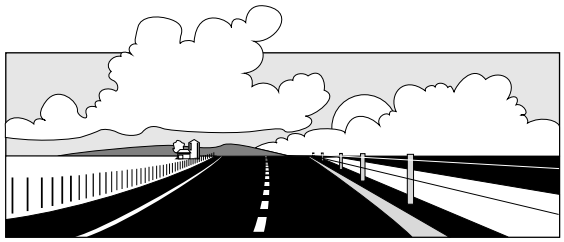
When #begdef boilerplate is encountered, subsequent lines up to #enddef can be placed in a list, which in turn is put in a table of definitions keyed by the names of macros.

When >boilerplate is encountered, the value of input (in the form suggested above) can be changed to the list associated with boilerplate, and then changed back to the former stream when the end of the list is encountered. The former stream may be a file or a list (in the case of nested macro invocation).

We'll leave the details to you. You'll probably want a stack for streams, especially if nested macro invocations are allowed. If you want nested macro *definitions*, you'll need to do something fancier, as you will if you want arguments to macro invocations.

You might think of other possibilities — element generation apply to sets, records, and strings also.

Perhaps more interesting is finding a way to generalize the technique to use procedures and other expressions that generate values. Co-expressions may prove useful here.



What's Coming Up

As promised, we'll continue the series of articles on the Icon compiler, dealing first with optimizations related to types.

We'll also have an article on evaluating strings that represent Icon expressions and an article on how Icon allocates space for strings.

In the longer range we plan to have detailed case studies of programs and we'll include an occasional exercise or two for you to check your programming skill.

Incidentally, we welcome suggestions for topics for future issues of the Analyst. We take suggestions seriously and want to know what you do and don't like about the Analyst. We try to stay several issues ahead in our publication schedule, however, so we can't promise to respond promptly to your suggestions.

Tell a Friend

If you know persons who you think might be interested in the Analyst, please tell them about it. You don't have to lend them one of your copies — we'll be glad to send them a free sample copy. Just give us postal mailing addresses.

In these days of electronic information transfer, it's easy to forget that some things don't go well that way. We only distribute the Analyst my postal mail: Too much is lost if it is converted to pure text, and the PostScript files from the desktop publishing system we use are too large for electronic transfer (not to mention copyright problems).