The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1993
Number 18

In this issue …

Lost Languages — Rebus … 1
Text in X-Icon … 4
Anatomy of a Program … 8
From the Wizards … 12
Subscription Renewal … 12
What's Coming Up … 12

Lost Languages — Rebus

In the last issue of the Analyst, we described SL5, the precursor to Icon. In this article, we'll describe another, but quite different "lost language" — Rebus.

Rebus is a hybrid of SNOBOL4 and Icon. It has a syntax that is similar to Icon's but with the semantics of SNOBOL4.

Background

SNOBOL4 is essentially two languages packaged together. One of these languages is imperative ("Do this, do that.") in which control flow is sequential and computations are of an ordinary nature. We'll call this language L. The other language, for pattern matching, is more declarative in nature ("Here's what it looks like, find something."). We'll call this language P. P was novel when it was designed in the 1960s and still is unusual compared to most other programming languages. If you know SNOBOL4, you'll understand what we mean. If you don't, the following quotation will give you some hints [1]:

> Programming languages such as Pascal, Basic, C, and assembler, with their if … then … else, repeat … while mentality, are serial, sequential, ploddingly left-brained. [Pattern matching in] SNOBOL4 seems to be parallel, associative, intuitive, right-brained. This is the crux of the matter. Certainly we use our left brain for problem solving, but how many of us think exclusively in terms of if … then … else? Imagination, creativity, great leaps of conception seem to originate in the inductive, parallel-functioning right brain. And this is precisely where SNOBOL4's pattern-matching abilities lie.

SNOBOL4 still is in widespread use, and most SNOBOL4 enthusiasts point to pattern matching as its most attractive feature. Yet SNOBOL4 has several problems that diminish its usefulness and turn away potential users.

One of the problems with SNOBOL4 is fundamental and lies in the "marriage" of the two dissimilar languages, L and P. You have to think differently to program in P than you do to program in L. Shifting from one language to another is distracting and leads to confusion and errors. Icon tried to solve this problem by integrating string scanning into an imperative context, but some SNOBOL4 programmers who also know Icon still tout the power of pattern matching with its declarative flavor.

Another problem with SNOBOL4 lies in its syntax, which relies on conditional gotos and labels to control program flow in L. It has none of the conventional control structures that are so useful for formulating computation and guiding logical thinking.

SNOBOL4 to Rebus

It is this second problem that Rebus attempts to solve. The idea is simple: Provide a syntax similar to Icon's for a better structured L and translate Rebus programs into SNOBOL4. This leaves the pattern-matching powers of P intact (and does not solve the L-P problem). Thus, Rebus has all the power of SNOBOL4, but it is packaged differently.

For the most part, the L component of Rebus is like Icon, while its P component is like SNOBOL4, with a few changes for syntactic consistency. Some

aspects of the L component of SNOBOL4 were carried over to Rebus. For example, Rebus uses SNOBOL4-style input and output, in which assignment to the variable output results in the value being written, and use of value of input causes a line to be read to supply that value.

There are lots of details that aren't important unless you're writing Rebus programs [2]. A general idea of what's involved can be seen in Figures 1-3, which show the same program written in SNOBOL4, Rebus, and Icon. To avoid a false impression of differences, the SNOBOL4 program is written with lowercase letters, although standard SNOBOL4 requires uppercase ones.

At first glance, the three programs may not appear to be very different. Compare the SNOBOL4 version to the Rebus version. Even if you don't know SNOBOL4, you can see the use of conditional gotos and labels in the SNOBOL4 version, while the Rebus version uses conventional control structures. The Rebus version is somewhat shorter than the SNOBOL4 version, partly because of other syntactic conveniences that Rebus provides. The Icon version, as expected, looks much like the Rebus version.

Although Rebus doesn't attempt to address the L-P problem, it's worth noting how patterns are used in the SNOBOL4 and Rebus versions of the program, as compared to Icon's string scanning. In the SNOBOL4 and Rebus versions, wpat is assigned a pattern value. This pattern is a small P program that is created during program initialization and subsequently is used to analyze the string text, find a word in it, and delete all but the remaining portion. The point is that wpat is created in one place and subsequently used in another. In the Icon version of this program, conventional string scanning is used, in which scanning expressions operate directly on text.

```
        letter = "abcdefghijklmnopqrstuvwxyz"
+                "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        wpat = break(letter) span(letter) . word
        count = table()
read    text = input                :f(sort)
        findw   text wpat =         :f(read)
        count[word] = count[word] + 1
+                                   :(findw)
sort    result = sort(count)        :f(nowords)
        output = "Word count:"
        output =
        i = 0
print   i = i + 1
        output = rpad(result[i,1],15)
+       lpad(result[i,2],4)         :s(print)f(end)
nowords output = "There are no words"
end
```

Figure 1. SNOBOL4 Word-Counting Program

```
function main()
   letter := &lcase || &ucase
   wpat := break(letter) & span(letter) . word
   count := table()
   while text := input do
     while text ?– wpat do
       count[word] +:= 1
   if result := sort(count) then {
     output := "Word count:"
     output := " "
     i := 0
     repeat output := rpad(
       result[i +:= 1,1],15) || lpad(result[i,2],4)
     }
   else output := "There are no words"
end
```

Figure 2. Rebus Word-Counting Program

```
procedure main()
   letter := &lcase ++ &ucase
   count := table(0)
   while text := read() do
     text ? while tab(upto(letter)) do
       count[tab(many(letter))] +:= 1
   result := sort(count, 3)
   if *result > 0 then {
     write("Word count:\n")
     while write(left(get(result), 15),
       right(get(result),4))
     }
   else write("There are no words")
end
```

Figure 3. Icon Word-Counting Program

Discussion

Why didn't Rebus take over, at least in the SNOBOL4 programming community? Part of the reason lies in the fact that Rebus doesn't offer *that* much of an advantage over SNOBOL4. And to use Rebus, a SNOBOL4 programmer has to make an

investment in learning a new syntax and converting existing programs.

From an Icon programmer's point of view, there's not only learning a new syntax, but also the semantics of SNOBOL4. Icon also has many features that SNOBOL4 lacks, including a more powerful expression-evaluation mechanism and a more extensive function repertoire.

Another strike against Rebus lies in the way it is implemented. A preprocessor (actually an Icon variant translator [3]) translates Rebus source code to SNOBOL4 source code, which then runs under SNOBOL4. Using a preprocessor makes the implementation easy, but it adds another "layer". The layer can be hidden in a script, but it's still there, and both the preprocessor and SNOBOL4 are needed to run Rebus.

Preprocessors also create problems in relating run-time errors to the source program, both in nature and in location. The preprocessor for Rebus goes to some lengths to handle the location problem, as is illustrated in Figure 4, which shows the actual SNOBOL4 program produced from the Rebus program in Figure 2 (white space has been adjusted to improve readability). Errors are, however, still reported in SNOBOL4 terms, which can be disconcerting for a Rebus programmer. And, as you can see, there's some overhead for error handling.

```
          DEFINE("MAIN()","NOMAIN_")
          line_ = 1;   DEFINE("MAIN()")                      :(L.9)
MAIN
          line_ = 2;   LETTER = (lcase_ ucase_)
          line_ = 3;   WPAT = (BREAK(LETTER) SPAN(LETTER) . WORD)
          line_ = 4;   COUNT = TABLE("")
L.1
          line_ = 5;   TEXT = INPUT                          :F(L.2)
L.3
          line_ = 6;   TEXT WPAT = ""                        :F(L.4)
          line_ = 7;   COUNT[WORD] = COUNT[WORD] + 1         :(L.3)
L.4                                                          :(L.1)
L.2
          line_ = 8;   RESULT = SORT(COUNT)                  :F(L.7)
          line_ = 9;   OUTPUT = "word count"
          line_ = 10;  OUTPUT = ""
          line_ = 11;  I = 0
L.5
          line_ = 13;  I = I + 1
          line_ = 15;  OUTPUT = (RPAD(RESULT[I,1],15) RESULT[I,2])  :S(L.5)
L.6                                                          :(L.8)
L.7
          line_ = 17;  OUTPUT = "There are no words"
L.8
          line_ = 18;  MAIN = ""                             :(RETURN)
L.9
          line_ = 0
          lcase_ = "abcdefghijklmnopqrstuvwxyz"
          ucase_ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
          &TRACE = 1
          DEFINE("ETRACE_(i)")
          &ERRLIMIT = 1
          TRACE("ERRTYPE","KEYWORD",,"ETRACE_")
          MAIN()                                             :(END)
ETRACE_
          stno_ = &LASTNO
          errtext_ = TRIM(REPLACE(&ERRTEXT,ucase_,lcase_))
          OUTPUT = "***** Error Termination *****"
          OUTPUT = "   Rebus source line number: " line_
          OUTPUT = "   SNOBOL4 statement number: " stno_
          OUTPUT = "   SNOBOL4 error number: " &ERRTYPE
          OUTPUT = '   SNOBOL4 error text: "' errtext_ '"'   :(END)
NOMAIN_   OUTPUT = "***** No main procedure *****"           :(END)
END
```

Figure 4. Output of Rebus Preprocessor

Rebus also was never given extensive support or much publicity. These days even an outstandingly good programming language doesn't stand much of a chance without promotion.

Conclusions

If Rebus had a "native" implementation (a *major* undertaking) and had been aggressively "marketed", it might have been developed a fol-

lowing. But probably not. Rebus is just another of those many programming languages that have a few good ideas but never "made it".

Actually, Rebus is not dead. Although the Icon Project decided it could not afford to support Rebus, Mark Emmer at Catspaw, Inc. developed an MS-DOS version that still is available, both from Catspaw and electronically from the Icon Project. Source code is available and it should be possible to get Rebus working again on other platforms.

Incidentally, the name Rebus was chosen for its meaning and is not an acronym. ICURYY!

References

1. *SNOBOL4+; The SNOBOL4 Language for the 8086/80886 Computer Family*, Mark B. Emmer, Catspaw, Inc., Salida, Colorado, 1984.

2. *Rebus — A SNOBOL4/Icon Hybrid*, Ralph E. Griswold, technical report TR 84-9, Department of Computer Science, The University of Arizona, 1984.

3. "Variant Translators", The Icon Analyst 7, pp. 2-5.

Text in X-Icon

When you think of X-Icon, you're likely to think of drawing and images, not text. Text is, however, an important aspect of many graphic applications. It is fundamental to word processing and desktop publishing, and some text appears in almost all graphic applications.

The File Model of Windows

In order to understand how to use text in windows, it's necessary to understand how X-Icon treats windows. A window, created by open() with the mode "x", is an extension of Icon's file data type. (Windows have type "window", not "file", so that they can be easily distinguished when programming.) When a window is created, it is opened for both reading and writing. Icon's functions for reading and writing can be used for a window just as if it were an ordinary file. An example is:

```
&window := open("text", "x") |
  stop("*** cannot open window")

while write(&window, read())
```

which fills the window from standard input.

Output written to a window scrolls automatically, just as if the window were a typical terminal text screen. When a window is full, text flows off the top to make room for more at the bottom.

Text also can be read from a window, as illustrated by

```
repeat {
  writes(&window, "command? ")
  case read(&window) of {
    "quit":       exit()
    "continue":   break
    "erase":      XEraseArea()
    }
  }
```

Note the use of writes() so that the text entered by the user is on the same line as "command? " and follows it.

The window must be the first argument when reading and writing to a window, since the default first argument of output is &output and the default first argument for reading is &input.

Positioning Text

X-Icon maintains a position at which text is written. The horizontal text position is advanced as characters are written, and a newline character causes the vertical position to advance. Text position can be specified in terms of rows and columns and is one-based. That is, a character in the upper-left corner of a window is at row 1 and column 1. This is the position at which the first character is written after a window is opened if the text position is not changed. Rows are counted from top to bottom in a window, and columns are counted from left to right.

The function XGotoRC(r, c) sets the window location to row r and column c and can be used to set a specific row-column position. For example, XGotoRC(1,1) sets the location to the upper-left corner of the window, and text written subsequently starts there.

The function XGotoXY(x, y) can be used to set the text position to a specific x-y pixel location. Pixel positioning can be useful in placing text more precisely than row-column positioning allows. Note that the arguments in XGotoRC() and XGotoXY() specify horizontal and vertical positions in different orders.

Warning: The keywords &x, &y, &row, and &col refer to the location at which events occur.

Changing the values of these keywords does not change the location at which text is written.

There is a text cursor that indicates the current window location at which text will be written. This cursor normally is invisible, but can be made visible by setting the attribute cursor to on, either when a window is opened or by XAttrib():

```
XAttrib("cursor=on")
```

The text cursor is an underscore and does not blink. Consequently, it may be difficult to locate in a large window filled with text or if the text contains underscores.

Fonts

One advantage of using X-Icon in textual applications is the availability of various text fonts. A font, as defined in X, is a set of characters in a particular style and size. The style distinguishes the appearance of the characters and includes not only the general characteristics of the characters (their "face" or "family") but also whether they are normal (roman), italic, bold, and so forth.

In this article, most of the text is in a font from the Palatino family, while program material is in a font in the Frutiger family. The differences in the appearances of the two fonts allow program material to be easily distinguished from the body of the text.

Fonts are complicated; immensely complicated. There are thousands of them, including ones in various languages and some consisting of symbols and "pi" characters for special purposes. Esthetics play a very important part in font usage. This is somewhat diminished in graphic applications, since the resolution of computer displays is too low to get really attractive results. In any event, you don't have to be a font expert to use fonts in X-Icon in useful and attractive ways.

Fonts can be divided into two general classes: monospaced ones, like Courier, in which every character has the same width, and proportionally spaced fonts like Palatino, in which characters have different widths according to their visual width (an i being narrower than, for example, an o).

Monospaced fonts are holdovers from typewriters, line printers, and computer terminals, for which the printing technology made fixed spacing necessary. Monospaced fonts have two distinct advantages: the characters line up in columns and layout is simple.

Proportionally spaced fonts are more visually attractive and easier to read than monospaced fonts, and are, of course, used for most printed material. They also allow more characters to be displayed in a given space than monospaced fonts do.

In X, fonts have string names. Some font names are simple, like fixed, which is a utilitarian monospaced font of an average size. In general, however, X identifies fonts with "names" that contain many dash-prefixed fields, such as the vendor, the family, various style attributes, the pixel size, and a few other things that are of interest only to specialists [1]. An example is shown in Figure 1.

Such font names are pretty daunting. Fortunately you usually don't have to specify fonts precisely this way; wild cards (*) can be used for fields you don't care about. For example,

```
–*–*–*–*–*–*–10–*–*–*–*–*–*–*
```

specifies fonts that are 10 pixels high.

There also are some short font names in addition to "fixed". Still, font specification in X is painful. It may (or may not) help you to know that the X Consortium had reasons for using this method of font identification [1]. In any event, you're stuck with it, although there is some help for font selection. See the procedure XBestFont() in xbfont.icn in the Icon program library, and run tryfont to see how it works.

If font names weren't bad enough, fonts reside on the X server and vary from server to server. The same font may have different names on different servers and more than one name on a specific server. Workstation servers usually have several hundred different fonts, but the specific ones vary from platform to platform. X-terminal servers may only have a few fonts. The result is that you can't count on what fonts will be available when your X-

```
–adobe–new century schoolbook–bold–i–normal––14–100–100–100–p–88–iso8859–1
```

Figure 1. An X Font Name

Icon program runs, unless its always runs on the same server. The one thing you can count on is that a server will have at least one font. We think (but are not certain) that the font fixed is always available (and X-Icon won't run if it isn't).

A font can be specified when a window is opened, as in

```
&window := open("text", "x",
  "font=lucidasans–12") |
    stop("*** cannot open window")
```

If no font is specified, fixed is provided. If a font is specified but not available, open() fails.

The current font can be determined by using XFont(), as in

```
write("The font is ", XFont())
```

or set by supplying a font name, as in

```
XFont("fixed")
```

XFont(s) fails if not s is not the name of an available font.

Font Characteristics

For many purposes, it is sufficient to pick a font and write text just as you would to a terminal. For some purposes, however, additional characteristics of fonts may be useful. Figure 2 shows the font-dependent attributes that are associated with characters.

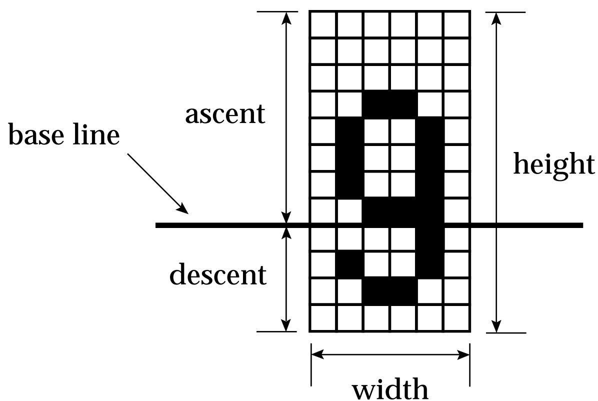


Figure 2. Font Attributes

The base line is the line on which a character "sits" — the y coordinate of the current text position. The ascent portion is above the base line, while the descent portion is below. In most fonts, only a few characters, such as g, p, and y, actually go below the base line.

As shown in Figure 2, there typically is space above, below, and to the sides of the pixels that comprise a character. This prevents characters from running together. How much space there is and where it is varies from font to font. Many fonts have all the space between characters at the right, so that a character written in column 1 touches the left edge of the window, which is visually unattractive.

The height of a font is no guarantee of how tall characters are. For example, a T in Palatino is considerably taller than a T in Zapf Chancery in the same size (*T*).

The term *leading* refers to the distance between the base lines of text written on successive lines. Writing a newline character starts a new line and advances the horizontal position for text by the amount of the leading. The leading associated with a font normally is the same as the font height (the line spacing having been considered in the font design).

The various characteristics of a font are available in attributes that are set when the font is selected:

fheight	height of the font
fwidth	width of characters in the font
ascent	extent above the base line
descent	extent below the base line
leading	distance between base lines

All values are in pixels. The first four attributes are properties of the font and cannot be changed.

In the case of a proportionally spaced font, fwidth is the width of the widest character, which normally is M or W. Columns for proportionally spaced fonts are based on fwidth, although, of course, characters in proportionally spaced fonts usually do not line up in columns.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

RBBS: (602) 621-2283

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

The leading can be changed to adjust the space between lines. For example

```
XAttrib("leading=" || 2 * XAttrib("fheight"))
```

produces “double spacing”.

Text Width

For a monospaced font, the width of a string when written is just the character width times the number of characters in the string. For a proportionally spaced font, however, the width of a string when written obviously is more complicated.

The function XTextWidth(s) returns the width (in pixels) of the string s in the current font.

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!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

© 1993 by Madge T. Griswold and Ralph E. Griswold
All rights reserved.

Drawing Strings

In addition to writing text to a window, you also can draw strings using

```
XDrawString(x, y, s)
```

which draws the string s starting at the specified pixel location. XDrawString() only draws the characters of the foreground pixels, not the background color that normally fills the “space” around text when it is written.

Strings that are drawn look the same as strings that are written. The primary reason for drawing a string rather than writing it is to take advantage of drawing attributes, and in particular to be able to erase text. If the drawop attribute is "reverse", as in

```
XAttrib("drawop=reverse")
```

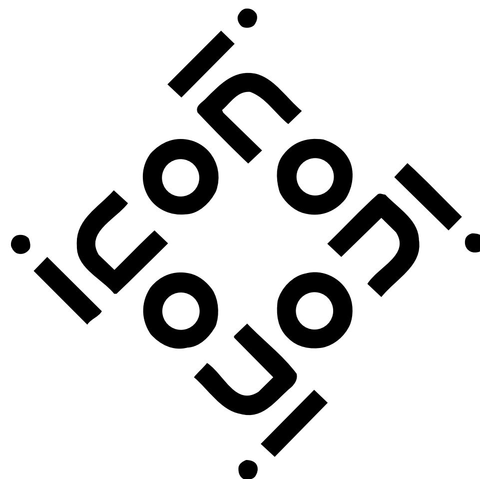
then a string drawn a second time at the same position erases the first one. For example,

```
every x := 20 to 100 do {
    XDrawString(x, x, "Hello!")
    XFlush()
    delay(1)
    XDrawString(x, x, "Hello!")
    }
XDrawString(x, x, "Hello!")
```

moves "Hello!" diagonally on the window, leaving a final copy at the end.

Reference

1. *X Logical Font Description Conventions; Version 1.4, MIT X Consortium Standard, X Version 11, Release 5*, Jim Flowers, Digital Equipment Corporation, 1992.



Anatomy of a Program — Timing Icon Expressions

In the first two issues of the Analyst, we described a program, empg ("expression measurement program generator"), for timing Icon expressions.

This program has evolved somewhat since then and we've picked it for a case study in program design. This also gives us a change to correct some earlier errors.

Designing this program presents interesting problems. One is figuring out how to get correct and accurate timings. The program also writes another program, which raises several issues.

The Program Model

The first question is one of basic design. A person wanting to know how much time it takes to evaluate various expressions probably would like to be able to run the timing program interactively, typing in expressions and getting the results.

There's no way to do that directly in Icon, since Icon does not have facilities for run-time translation of Icon code. On platforms that support the system() function, it's possible to give the appearance of run-time translation by writing a program to a file and then translating and executing that file. Except on the fastest platforms, however, the overhead for this on a per-expression basis makes the processing intolerably slow. Furthermore, the idea of entering expressions and getting timings interactively turns out not to be so appealing anyway; it takes too long to do the actual timings for reasons explained in the next section.

Consequently, empg is designed to read a sequence of expressions from standard input — either from a file or from the keyboard — and create a program that, when run, times the expressions and writes out the results. Thus, there are two steps: (1) running empg and (2) translating and running the program that empg produces.

Measuring Execution Time

On platforms with multiprocessing operating systems and particularly with multiple users, the value reported by the system clock for an individual process may vary, depending on the computational load or even on the nature of the computation. In our Sparc environment, we have observed timing differences of as much as 20%. There's not much you can do about this except to take several timings and average them. And to realize that timing figures are not going to be precise.

There are, however, some things to consider to avoid misleading or simply incorrect results. Since the clock resolution in most computers is low compared to the time it takes to execute a typical Icon expression, getting anything resembling accurate timing dictates that an expression be executed many times and an average time computed. On a typical platform, it may be necessary to execute an expression as many as 10,000 times in order to get a meaningful result. Consequently, empg needs to write a loop that evaluates an expression repeatedly.

In Icon, the natural way to do this is with an every loop, such as

```
stime := &time
every 1 to limit do expr
write(real(&time – stime) / limit)
```

where *expr* is the expression to be timed. (Note that &time gives elapsed CPU time in millisecods — the time spent in computation — not wall-clock time.)

Assuming that limit is large enough to avoid misleading results from low clock resolution and that the time needed to assign the starting time and take the difference with the time after the loop ends is insignificant, this looks like a good start. However, if *expr* is a simple expression, a significant part of the elapsed time may be in the overhead for the loop. So to get reasonable results, you might compute the loop overhead as follows:

```
stime := &time
every 1 to limit
overhead := (&time – stime) / limit
```

That's not bad for a first attempt, but you might wonder if omitting the do clause makes the loop faster. Indeed it does. And for timing simple expressions, computing loop overhead in this way gives misleading results. We'll come back to how you might find this out. For now we'll just tell you that it's the case.

What to do? One possibility is to compute the overhead with the simplest expression that you can imagine and figure that the results for other expressions will just be off by a little. One possibility is

```
stime := &time
every 1 to limit do &null
overhead := (&time – stime) / limit
```

This assumes that &null is fast (which it is; more on that later). But is &null faster than, say, a simple variable reference? And suppose you want to time some other fast expression or even &null. You could try other simple expressions in place of &null, but there's always the same bind; you can't hope to get accurate timings for simple expressions by this method.

To do any better, you need to look at the code the produced by icont, the translator for the Icon interpreter. (The optimizing Icon compiler is another matter altogether; we'll stick to the interpreter here.)

It's easy to get the code produced by icont and it doesn't take a lot of knowledge to discover what's needed for our purposes. Just run icont with its –c option to preserve the intermediate "ucode" files it produces [1]. If you do this, you'll see that the ucode generated for

```
every 1 to limit
```

looks something like this (the details of ucode may vary depending on the version of Icon you're using, but the essential features are the same for all versions):

```
mark0
pnull
int     0
var     0
push1
line    2
toby
pop
efail
```

while the ucode for

```
every 1 to limit do expr
```

looks like this:

```
mark0
pnull
int     0
var     0
push1
line    2
toby
pop
mark0
<code for expr>
unmark
efail
```

This shows why the omission of the do clause makes a difference. (It's not necessary to know what the mark0 and unmark do; it's enough to know there are extra instructions.)

Incidentally, depending on how you arrange the lines of your source code, you may get more or fewer line instructions. This actually is not a problem for timing, since line instructions do not end up in the "icode" file that is produced by Icon's linker, which is what is actually "executed".

At this point, you may be wondering just how to compute the overhead so that the timing for any expression is correct. You might think of

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

for overhead computation and

```
every 1 to limit do {
  &null
  expr
  }
```

for timing *expr*. The braces are unnecessary in the first case and are included only to emphasize the relationship between the two expressions. In case you are wondering, braces do not produce any ucode.

It *looks* like the difference in the code for these two loops should be just the code for *expr*. But if you compare the ucode for the do clauses in the first and second loops, you'll see that there are some extra instructions in the second loop around the instructions for the keyword. These have to do with bounded expressions, which prevent control backtracking from one expression in a compound expression into a former expression:

First do *clause:*

```
mark0
keywd 34
unmark
efail
```

Second do *clause:*

```
mark0
mark   L4
```

```
        keywd 34
        unmark
    lab L4
        <code for expr>
        unmark
        efail
```

Is there a way to get rid of the extra ucode? Well, control backtracking is not prevented in conjunction. Indeed, the ucode for an expression like

```
expr1 & expr2
```

is

```
<code for expr1>
    pop
<code for expr2>
```

where the pop instruction merely gets rid of the result produced by *expr1*. Surely the time it takes to execute the pop instruction can't be significant.

This suggests that the loop overhead should be determined by

```
every 1 to limit do                    e1
  &null
```

and the timing of *expr* should be done by

```
every 1 to limit do                    e2
  &null & expr
```

If we use the notation t(*e*) to denote the time it takes to execute expression *e*, then we have

$$t(e1) = \Theta_{limit} + t(\&null)$$
$$t(e2) = \Theta_{limit} + t(\&null) + t(pop) + t(expr)$$

where Θ_{limit} is the time for executing the loop limit times. Consequently,

$$t(e2) - t(e1) = t(expr) + t(pop)$$

As suggested above, you might think that t(pop) could be ignored; all it does is pop a value off the interpreter's evaluation stack, which can't take much time compared to the time for an entire Icon expression. Actually, t(pop) is not insignificant. The problem is that although popping a value off the interpreter's evaluation stack is fast, pop is a virtual machine instruction that must be fetched, decoded, and executed in software, much the same way real machine instructions are fetched, decoded, and executed in hardware [2]. Even though the execution of pop itself is fast, the fetch and decode operations take enough time that they need to be considered.

In fact, if you look at the implementation of the Icon interpreter closely, you'll find that the code for &null is treated specially. At the icode level, it doesn't involve a keyword look-up like most other keywords — it just pushes a null value on the evaluation stack. Considering the overhead for fetching and decoding, the time to execute the Icon expression &null is only a little more than the time to execute the virtual-machine instruction pop. That is,

$$t(pop) \cong t(\&null)$$

Going back to the loop *e2*, consider timing &null:

```
every 1 to limit do                    e3
  &null & &null
```

so that

$$t(e3) = \Theta_{limit} + 2 \times t(\&null) + t(pop)$$

Consequently,

$$t(e3) - t(e1) = t(\&null) + t(pop) \cong 2 \times t(pop)$$

or

$$t(pop) \cong (t(e3) - t(e1)) \;/\; 2$$

and hence, with some simple algebra

$$t(expr) \cong t(e2) - (t(e1) + t(e3)) \;/\; 2$$

Program Design

Now that we know how to get a good approximation to the time required to evaluate an expression, it's time to deal with the problem of writing the program to process the expressions to be timed and actually producing the timing code.

We're now into the design of empg and questions about its functionality. A user of empg is certainly going to want to time several expressions in a "batch".

A simple and natural format for input is one expression per line. (If we want to generalize this later to allow multi-line expressions, that shouldn't be hard.) Consequently, at an elementary level, empg will process input lines and output the Icon code for timing each one.

If you think about this simple model for a moment, you'll see more is needed. For example, there needs to be a way to initialize variables that are used in expressions to be timed. Suppose, for

example, you want to time list subscripting, as in

```
L := list(10)
L[1]
```

It's possible, of course, to just time both expressions in loops — a correct value for L will be left over after the first expression is executed in a loop. But this is very time-consuming and there are some kinds of initializations that cannot be done if executed repeatedly (can you think of examples?).

This suggests that empg should provide a way to evaluate specified expressions just once as opposed to evaluating all expressions in loops. Consequently, there needs to be some way to tell the two cases apart.

There are many possible ways to handle this, like a command syntax or an initialization section in the input to empg. We'll use something simple, if inelegant, and let the first character of a line serve as an indicator of what kind of code is to be generated for the rest of the line. Using the first character makes the program simpler, since string scanning in Icon proceeds naturally from left to right and consequently it's known what to do with the rest of the line before it's encountered.

It also would be convenient for the user of empg not to have to do anything special about the case that's likely to be the most common: timing expressions in a loop. This suggests identifying expressions to be evaluated only once by a character that cannot occur at the beginning of an Icon expression. One such character is the colon, so we'll say, somewhat arbitrarily, that an input line that begins with a colon indicates that the remainder of the line is an expression that is to be evaluated only once. For the example above, the input then would be

```
:L := list(10)
L[1]
```

It's not enough to provide for expressions that are to be evaluated only once. Suppose you want to use a procedure or a record in an expression to be timed. A facility for handling declarations is needed. That's simple enough. We'll just pick another character that can't occur at the beginning of an expression to identify lines of input that comprise declarations.

If you look at Icon's syntax closely, you'll see there aren't many good choices — we've almost boxed ourselves in by the syntactic device we've picked. The dollar sign ($) looks like a good choice, but it's used on EBCDIC platforms to begin two-character sequences that represent characters that are not available on many IBM terminals. And, starting with Version 8.10 of Icon, the dollar sign is used by the preprocessor. Another possibility is the grave accent(‘), but it's barely visible. There is one other character, however — the percent sign (%), and we'll use it with hopes that yet another line differentiator is not needed. Or that someone does not decide to add a prefix % operator to Icon.

Actually, the situation is not all that bad. It would be easy enough for a user of empg to work around the first-character restriction by putting a blank in front of an expression that is to be timed in a loop.

Using %, a declaration in empg's input looks like this:

```
%procedure max(i, j)
%   return (i < j) | i
%end
```

We deliberately ignored the pound sign (#), even though it cannot appear at the beginning of an Icon expression, since it might be confused with a comment. And the user of empg might like to include comments in the input files. Thus, this character has another natural use. Note that a comment in the output of empg can be obtained by using %#, as in

```
%# max(i, j) returns the maximum of i and j.
```

since, as you would expect, lines beginning with a % will be written out by empg with just the % removed.

Next Time

It's taken us four pages just to get this far — and we've not even produced a single line of code for empg itself. We'll put off the remainder of this case study until the next issue of the Analyst, where we'll consider how the program actually is written. We'll give a complete listing of the program there.

References

1. "An Imaginary Icon Computer", Icon Analyst 8, pp. 2-6.

2. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986, pp. 110-129, 264-278.

From the Wizards

It's been a while since we've seen a use of Icon that deserves the "wizard" label. Here's one from K'vin D'Vries.

He wanted to determine whether a value was less than 0, greater to zero, or equal to zero.

The obvious method is

```
if i < 0 then expr1
else if i > 0 then expr2
else expr3
```

But he thought a case expression would be more compact and clearer. The temptation is to write

```
case i of {
  <0:   expr1
  >0:   expr2
  0:    expr3
  }
```

This, however, is syntactically incorrect, since <0 and >0 aren't complete expressions.

K'vin observed, however, that the following will do:

```
case i of {
  0 > i:   expr1
  0 < i:   expr2
  0:       expr3
  }
```

This works because 0 > i and 0 < i, like all Icon comparison operations, produce the value of their right operands if they succeed.

You might argue that this is tricky and that it's easy to make a mistake by writing, for example, i < 0 instead of 0 > i. Of course, that's true, but experienced Icon programmers constantly make use of the value returned by comparison operators, as in i < j < k, which succeeds if and only if j is strictly between i and k.

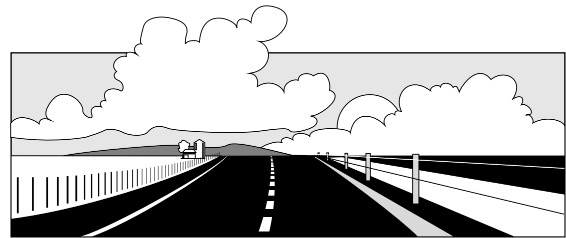
And we think the "D'Vries Device" is so clever that it deserves your consideration. Think of all the similar things you can do.

Subscription Renewal

For many of you, this is the last issue or next-to-last issue in your present subscription to the Analyst. If so, you'll find a subscription renewal form in the center of this issue.

With the next issue, we'll be starting the fourth year of the Analyst. We have lots of interesting articles coming up. In addition to those mentioned for the next issue in **What's Coming Up** below, we'll continue the series of articles describing how to program in X-Icon: manipulating images, dealing with color, building user interfaces, and so forth. We also have articles in the works on string invocation, programmer-defined control operations, recursive-descent parsing, prototyping, debugging, using random numbers, names and variables, and writing program monitors and visualization tools.

Renew now so that you won't miss an issue. Your prompt renewal also helps us manage our resources.



What's Coming Up

In the next issue of the Analyst, we'll describe the last "lost language" in our series — Seque, in which sequences are data objects.

We'll also continue our series on X-Icon with an article on handling events — mouse click and text entered in windows. And we'll finish the anatomy of empg.

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.