# The I con Analyst

*In-Depth Coverage of the Icon Programming Language*

June 1998
Number 48


### In this issue …


Line Termination .......................................... 1
Character Patterns ........................................ 3
Assault on Mount Versum .......................... 7
File-System Navigation Using VIB ........ 10
Programming Tips ..................................... 15
Subscription Renewal ............................... 16
What's Coming Up .................................... 16


## Line Termination

### Text and Binary Files

The files that can be read and written by Icon fall into two general categories: text files and binary files. This division is more a matter of how a file is used than any inherent property of a file.

Text files consist of lines that are followed by *line terminators*, while there is no concept of lines in a binary file. Line terminators are just characters reserved for the purpose of terminating lines in text files.

The lines in text files can be arbitrarily long, although most are intended for printing or display on a monitor and usually are of modest length. Some applications have limits on the lengths of lines they can read and write. Icon does not.

The function read() is intended for reading text files. It recognizes line terminators, discards them, and returns the characters up to line terminators. The last line of a file read in text mode by Icon need not have a terminator; Icon handles this as a special case. Be aware though, that if the last line of a text file does not have a terminator, some applications may discard the last line when reading.

The function reads() is intended for reading binary files. It does not recognize or discard line terminators (which may appear in binary files but have no intended special status), but instead reads a specified number of characters. For example,

```
reads(&input, 10)
```

reads and returns the next 10 characters from the standard input file. If there are fewer than the specified number of characters, a shorter string is returned unless no characters remain, in which case reads() fails.

In writing, write() appends a line terminator to the string it writes, while writes() does not. writes() is appropriate for writing binary data or portions of lines to which a line terminator eventually is added.

### Line Terminators

Reading and writing files would be considerably simpler if all platforms used the same characters for line terminators — or even the same number of characters for line terminators.

In the MS-DOS, and hence Windows, world, a line terminator is two characters: a "carriage return" (hex 0D) followed by a "linefeed" (hex 0A).

This convention was derived from manual typewriter technology, in which starting a new line required a carriage carrying a platen to be repositioned at the left — often with a disconcerting "clunk" — followed by notching up the platen to move the paper up.

This concept was carried over to computer terminals that originally, at least, were designed to behave much like typewriters. There being no physical "carriage" to return, the return character moved the cursor to the left of the current line and the linefeed character moved the cursor down one line to start a new one. (We wondered if we needed to explain this, but recalled that many college students don't know what a punched card is.)

It is, of course, not necessary to have two characters to accomplish the action of moving the cursor to the left and down a line; software can

easily interpret a single character to produce the desired effect. Note, however, that on such terminals, the two actions sometimes were required independently for some kinds of positioning.

Better designed operating systems long ago made the more intelligent decision to use a single character as a line terminator and let software provide the necessary interpretation.

The simplistic interpretation of a line terminator as two characters to represent two separate actions in MS-DOS and Windows platforms has created endless problems even for users of other operating systems — simply because file transfer between different operating systems is necessary and commonplace. Unfortunately, once such a decision is made, implemented, and widely used, there's no easy way back.

UNIX uses the linefeed character as a line terminator and calls it the newline character, but the Macintosh OS, which came later, chose to use the return character. While there are arguments as to which character is better, the ASCII standard specifies the linefeed character for this purpose. There seems to be a "not invented here" factor in Macintosh operating system design: being different only to be different. Granted, the Macintosh contributed many wonderful innovations to personal computer technology that later were lamely copied by its competitors. The choice of a non-standard line terminator is, however, an unnecessary complication, not an innovation.

Other recent operating systems use one of the three line terminators mentioned above. For example, the Amiga uses the linefeed, while the Atari (no longer common) uses return/linefeed.

Prior to Version 9.3.1 of Icon, the implementation of Icon for a specific platform used the line terminator native to that platform. With Version 9.3.1 [1], Icon recognizes all three line terminators, allowing text files written on one platform to be read by Icon on any platform.

C, the language in which Icon is implemented, originated in the UNIX environment. Naturally, its I/O libraries use the linefeed character when reading and writing text files. It's also common to find C programs in which the linefeed character is used explicitly with the expectation that it is the line terminator.

When C was ported to other platforms, the line-terminator problem had to be faced. Using different line terminators on different platforms would have invalidated programs that had "knowledge" of the line terminator — *and* that it was a single character. Instead, I/O libraries for different platforms were written to translate between native line terminators and the linefeed character when reading and writing text files. Thus, for example, a C program can write a linefeed character with the assurance it will appear as a native line terminator, regardless of the platform on which the program is run. *As long as the file is a text file.* Having a linefeed character translated into a return character or a return-linefeed pair would be disastrous for a file containing "binary" information.

It's worth noting that in the UNIX world, there is no real concept of text and binary files. It all depends on what's intended and how it is interpreted. Of course, programs expecting one or the other may malfunction if given inappropriate files. Some programs use *ad hoc* criteria to differentiate between text and binary data. A common method is to assume that a file containing any character with the 8th bit set is a binary file. These days, such characters provide an extended text character set that some software handles properly and, in fact, expects.

The Macintosh has from its conception used most of the 256 possible 8-bit values to represent text characters, making it the first widely available platform on which a variety of useful (even necessary) special characters could be directly represented.

## Translation Modes

To deal with the different interpretation of files, modes for reading and writing files were introduced. In the *translated mode*, native line terminators are translated coming in and going out. In the *untranslated mode*, they are not.

The mode is a property of how a file is opened, not of the file itself (there is nothing in a file that says "this is a text file" or "this is a binary file". In Icon, the mode can be specified in the second argument of open(), in which options for opening a file can be specified. The letters "t" and "u" stand for translated mode and untranslated mode, respectively. For example,

```
input := open("ledger.in", "u")
```

opens the file ledger.in for reading in text mode, while

```
output := open("new.db", "wu")
```

opens the file new.db for writing in untranslated mode. In the absence of a mode specification, the default is the translated mode.

It is particularly important to open a file in the correct mode running on MS-DOS/Windows. Note that when in UNIX, a mode specification has no effect, while on the Macintosh it does.

### A Complication

If there are complications when working across several operating systems, expect them to be for MS-DOS/Windows [1].

In MS-DOS, the end of a text file is indicated by the control-Z character (hex 1A) — even if that character actually is in the middle of a file. Our understanding is that in early versions of MS-DOS, this was the only way for the system to detect the end of a text file, although later versions of MS-DOS know how long the file is and accept the physical end of file as well as a control-Z. Perhaps someone with earlier experience with MS-DOS than we have can confirm or correct this impression.

It's easy enough to use control-Z only for its intended purpose when creating a text file, but a control-Z characters may — and often must — appear in binary files.

If a binary file is read in untranslated mode, this is not a problem. Simple enough, right? — just open the file in untranslated mode. Wrong. Files read from standard input are always translated; there's no way to prevent it.

MS-DOS/Windows users, not realizing this somewhat subtle problem, sometimes make the mistake of supplying a binary file as redirected input on the command line, as in

```
chrcount <setup.exe
```

intending, perhaps, to tabulate the characters in setup.exe. This almost always causes erroneous results, since input stops when a control-Z is encountered. In this case, if the error is detected at all, the file appears to have been truncated. This problem is the source of the most frequent single "trouble report" to the Icon Project.

Problems like these reside with the operating system and I/O libraries: They are beyond the reach of Icon.

If you need to read a binary file in untranslated mode, the only way to do it is by opening the file within the program. The program fileprnt.icn in the Icon program library provides an example of how to do this. Be aware, though, that programs written on platforms that don't have this problem are unlikely to have considered it in their design or even to have known it's a portability problem.

### Reference


1. "The Curse of DOS", *The Icon Newsletter* 54, pp. 1-2.


## Character Patterns

Patterns of digits often are evident in studying versum numbers and primes. In the case of versum numbers, patterns sometimes provide clues that are helpful in discovering relationships and basic properties. The 4-digit versum numbers that begin with the digits 2 through 9 provide an example:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2002 | 3003 | 4004 | 5005 | 6006 | 7007 | 8008 | 9009 |
| 2101 | 3102 | 4103 | 5104 | 6105 | 7106 | 8107 | 9108 |
| 2112 | 3113 | 4114 | 5115 | 6116 | 7117 | 8118 | 9119 |
| 2211 | 3212 | 4213 | 5214 | 6215 | 7216 | 8217 | 9218 |
| 2222 | 3223 | 4224 | 5225 | 6226 | 7227 | 8228 | 9229 |
| 2321 | 3322 | 4323 | 5324 | 6325 | 7326 | 8327 | 9328 |
| 2332 | 3333 | 4334 | 5335 | 6336 | 7337 | 8338 | 9339 |
| 2431 | 3432 | 4433 | 5434 | 6435 | 7436 | 8437 | 9438 |
| 2442 | 3443 | 4444 | 5445 | 6446 | 7447 | 8448 | 9449 |
| 2541 | 3542 | 4543 | 5544 | 6545 | 7546 | 8547 | 9548 |
| 2552 | 3553 | 4554 | 5555 | 6556 | 7557 | 8558 | 9559 |
| 2651 | 3652 | 4653 | 5654 | 6655 | 7656 | 8657 | 9658 |
| 2662 | 3663 | 4664 | 5665 | 6666 | 7667 | 8668 | 9669 |
| 2761 | 3762 | 4763 | 5764 | 6765 | 7766 | 8767 | 9768 |
| 2772 | 3773 | 4774 | 5775 | 6776 | 7777 | 8778 | 9779 |
| 2871 | 3872 | 4873 | 5874 | 6875 | 7876 | 8877 | 9878 |
| 2882 | 3883 | 4884 | 5885 | 6886 | 7887 | 8888 | 9889 |
| 2981 | 3982 | 4983 | 5984 | 6985 | 7986 | 8987 | 9988 |
| 2992 | 3993 | 4994 | 5995 | 6996 | 7997 | 8998 | 9999 |

If you look at these numbers in the right way, it's easy to see that adding 1001 to a number in a column gives the value of the number to its right. This property was discovered by looking for a pattern and then proving the general case [1].

Mathematicians commonly use this kind of approach to discovery but then publish only the result and its proof [2].

The human cognitive system often can sense the existence of patterns at a glance, but except in the simplest cases, it cannot easily characterize them or determine the underlying rules. This usu-

ally requires reasoning and tools — tools that range from making hand notations to using computer applications.

When working with versum numbers, we used various kinds of notational devices to describe patterns and relationships. Most of these didn't make it into the Analyst articles, but a few did, such as the notation $s_n$ to stand for *n* repetitions of the string *s*. This notation does two things: It shows a pattern and it abbreviates. The importance of abbreviation, even when a pattern is evident, is illustrated by this string:

```
00000000000000000000000000000000000000000000000000
```

It's obvious that it consists only of zeroes, but how many are there? For this, you need a "tool" of some kind, such making tick marks. (Without using any devices, and glancing at the line above only briefly, how many zeros would you guess there are?)

This inability to accurately perceive more a small number of things quickly is a well-known and fundamental limitation of the human cognitive system. This sometimes is called "Miller's 7 ± 2 Law" [3]. If you're not familiar with this aspect of the human cognitive system, try writing successively longer strings of (say) zeroes *without keeping track* and see how far you can get before you can't accurately tell how many there are at a glance.

Where patterns are important, it is necessary to have concepts about relevant patterns, notations for making them evident, and tools for discovering or creating patterns.

This article is the first in a series in which we will explore these topics. Although motivated by patterns in strings of digits, the problem is equally important in other kinds of string data and the techniques are equally applicable. Needless to say, we will soon come to programming.

## Character Encodings

The emphasis in this article is on *character* patterns and generally does not apply to situations in which the elements of interest are longer, such as words. The reason for this distinction will become clear before long.

In digit strings, the fundamental units are characters. However, there are many other situations in which single characters can be used to represent more complex entities.

This was the basis for describing geometric designs using Lindenmayer Systems and turtle graphics. For example, in the string

```
B := "F+FFff–FF+ffFFffFFff–FFFFFFFFF"
```

the character F indicates moving forward in the current direction while drawing, f is the same without drawing, + indicates a turn to the right, and – a turn to the left. There are evident patterns even in this short string.

Using characters to represent more complex entities has several advantages. Such representations are compact compared to the data they encode, and Icon's string processing operations can be used to analyze and manipulate the data. For example,

```
map(B, "+–", "–+")
```

produces a string with the directions of the turns reversed.

Other examples of this kind of encoding are given in the **Mappings and Labelings** chapter in the Icon language book [4]. Some are more obvious than others. Representing a standard deck of playing cards by 52 different characters is natural, even if the advantages of doing so are not clear. Other encodings, such as using characters to label the nodes of a directed graph, are less obvious. Consider the graph shown in Figure 1.

c e a b d f

**Figure 1. A Simple Directed Graph**

The arcs can be represented by a string of "from-to" character pairs:

```
arcs := "abbdbfcacbcedddf"
```

The direction of the arcs then can be reversed by

```
arcs := reverse(arcs)
```

There are two notable disadvantages to this kind of encoding: (1) To understand it, it is neces-

sary to know what the characters stand for and make the necessary translation, and (2) the number of different characters available is quite limited. We'll have more to say about the second problem in another article; for now, we'll assume that problem domain is such that encodings like these are appropriate. In the case of digit strings, the encoding is inherent and imminently natural (at least in base 10).

The next issue is what kinds of character patterns are appropriate. This, of course, depends on the problem domain. For analyzing versum numbers, two kinds of patterns stand out: palindromes and repetitions. In textile design (a synthetic process), these are important, although palindromes are treated differently (see the side bar). Interleavings and extensions to fit width constraints also are important patterns in textile design. In DNA analysis, other kinds of patterns are of interest, and so on.

We'll start with the familiar territory of digit strings and take up other problem domains in future articles.

## Patterns and Grammars

Three kinds of patterns seem useful in characterizing versum numbers and primes: specific substrings, repetitions, and reversals. We originally worked with palindromes, but eventually decided that reversals, which can be used to describe palindromes, were more useful. Consider, for example,

```
12345694560456045606328374560901456054321
```

This is far from being a palindrome, but the fact that its last six digits are the reversal of its first five might be interesting. Note also that the string 4560 is repeated three times near the beginning of the string and occurs at two other places. This might be important or irrelevant.

To capture such patterns, we need notation for them. Such notation needs to be linear and unambiguously representable in character strings. For repetitions and reversals, it needs to be a bracketing notation, since patterns can be nested.

For repetitions, we chose (*s*,*n*) to represent the string *s* replicated (concatenated) *n* times. Thus, the three repetitions of 4560 can be represented as (4560,3). When *n* is one, (*s*) suffices and provides "punctuation" that can be useful for making important strings more evident. For example, the string above might be shown as

```
1234569(4560,3)32837(4560)901(4560)54321
```

Note that the characters "(", ",", and ")" are meta-characters that are used to describe patterns and cannot be used for encoding data.

For reversals, we chose the notation <*s*> to denote the reversal of *s*. This adds the characters "<" and ">" to the list of meta-characters. The string now can be written as

```
(12345)69(4560,3)32837(4560)901(4560)<12345>
```

There are several things wrong with the pattern notation as it stands. Even in this simple example, instances of the "important" strings 12345 and 4560 and their relationships are not easy to identify. In more complicated cases, this problem can be intractable. In addition, the string with patterns encoded is longer than the original string, whereas brevity is an important consideration.

One way to overcome these problems is to use additional characters to represent the patterns. Thus, if A stands for 12345 and B stands for 4560, the encoding is considerably shortened and relationship between patterns is more obvious:

```
A69(B,3)32837(B)901(B)<A>
```

Now, however, the strings that A and B stand for are not contained in the string. Anticipating more such "labeled" patterns, as well as patterns within patterns, the obvious solution in Icon is to use a table. It can contain the "root" string, which we'll associate with the label ∗. It might start out with a single element:

```
pats := table()
pats["∗"] :=
"12345694560456045606328374560901456054321"
```

and acquire more elements as patterns are identified. With A and B added, the elements become

```
pats["∗"] := "A69(B,3)32837(B)901(B)<A>"
pats["A"] := "12345"
pats["B"] := "4560"
```

But this is just a representation of a (very simple) production grammar. It can, for example, be treated as an L-System [5-6]. This is handy, since we have programs for dealing with L-Systems. In the syntax used for these, an L-System for the pattern above is

```
axiom:*
* –> A69(B,3)32837(B)901(B)<A>
A –> 12345
B –> 4560
```

Although you can find the patterns used here by hand, that's not practical for long strings with a lot of structure. Another thing that may not be obvious in this simple example: There generally are many ways in which a string can be encoded in terms of patterns. To illustrate this, we'll shift from digits to letters to avoid any implication of numerical meaning. Consider this 360-character string:

```
bdabdabdabdabdabdabdabdabdababcbabcbabcb
abcbabcbabcbabcbabcbabcbababcbabcbabcbabc
babcbabcbabcbabcbabcbababcbabcbabcbabcbab
cbabcbabcbabcbabcbababcbabcbabcbabcbabcba
bcbabcbabcbabcbababcbabcbabcbabcbabcbabcb
abcbabcbabcbababcbabcbabcbabcbabcbabcbabc
babcbabcbababcbabcbabcbabcbabcbabcbabcbab
cbabcbababcbabcbabcbabcbabcbabcbabcbabcba
bcbabadbadbadbadbadbadbadbadbadba
```

This string is not contrived. It represents distances between consecutive versum numbers. We'll have an article on this subject in the next issue of the Analyst.

The string above contains only four different characters. This may seem obvious, but it's all too easy to miss a stray character that is similar in shape to another. And suppose the string was 3,600 characters long instead of 360.

It's clear at a glance that there is considerable structure in this string. In fact, there are many different ways of representing this string in terms of patterns. Here's one grammar:

```
axiom:*
* –> Ab<A>a
A –> BD
B –> (bda,9)
C –> (babc,9)
D –> (Cba,4)
```

Perhaps the most important feature (for our purposes) is that the root string is a palindrome, Ab<A>, followed by an a. A itself has structure, being composed of repetitions.

It may be easier to understand the structure by viewing it as a tree, as shown in Figure 2:

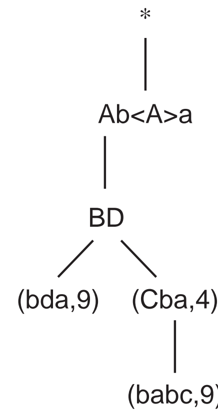



**Figure 2. A Grammar Tree**

The value of the grammatical representation can be seen by what the string would look like with just the pattern encodings:

```
(bda,9)((babc,9)ba,4)b<(bda,9)((babc,9)ba,4)>a
```

Here's a different grammar for same string:

```
axiom:*
* –> ACb<A>a
A –> (bda,9)
B –> (babc,9)
C –> (Bba,8)
```

This grammar is slightly shorter than the previous one, and the pattern string it describes is considerably shorter than the previous one:

```
((ba,9)a,8)(ba,9)(c,9)
```

What constitutes a good or possibly best grammar depends on the problem domain. We'll address this issue in a subsequent article.

## Next Time

As you might expect, we did not produce the grammars in this article by hand. In the next article in this series, we'll show how to find patterns and describe an application to help with the task.

## References

1. "Versum Numbers", Icon Analyst 35, pp. 5-11.

2. *Mathematics and Plausible Reasoning*, G. Polya, Princeton University Press, 1954.

## Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

3. “The Magic Number Seven, Plus or Minus Two”, G. A. Miller, *Psychological Review*, 63, 81-97.

4. *The Icon Programming Language*, 3rd edition, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, Inc., 1996, pp. 237-246.

5. “Anatomy of a Program — Lindenmayer Systems”, Icon Analyst 25, pp. 5-9.

6. “Anatomy of a Program — Lindenmayer Systems”, Icon Analyst 26, pp. 4-9.

## Assault on Mount Versum (continued)

In the last article on testing numbers for versumness [1], we described some initial steps we made to improve performance, but we stopped at the point where we decided to add memory for recording the results of computations.

### Adding Memory

Adding memory trades time for storage. In the case of vpred() as originally written, the two were out of balance — little storage was required, but the time required was unacceptably long.

The simplest form of memory would be a table subscripted by numbers with the corresponding values being the predecessors, if any. Unfortunately there is a complication: Some versum numbers have two predecessors [2].

We tried encoding the predecessors as strings. Although strings provide a compact representation, encoding and decoding the strings was too time consuming — in other words, not the right balance of time and memory for our situation.

Next we tried records whose fields contained the predecessors. That didn’t work: Although a versum number can have at most two predecessors, some intermediate values with leading zeros can have more. For example, with leading zeros allowed, as they must be in interior portions of versum numbers, the number 1098900 has three predecessors: 118089, 0119790, and 0549450.

Because of this, we decided to use lists. Here’s the way memory is used; some details and matters unrelated to adding memory are omitted:

```
# top-level procedure
procedure vpred(s)
  local t
  static done

  # initialize predecessor memory
  initial done := vpred_init()

  # reject numbers that can't be versum
      …

  # avoid buildup from successive calls
  vpred_done := copy(done)

  every t := vpred_(s) do
    if t[1] ~== "0" then suspend t

end

# internal workhorse
procedure vpred_(s)
  local t, v, p, u

 # check for zeros or empty string
      …

  v := vpred_done[s]  # get value if any

  if \v then {        # if there's a predecessor list
    suspend !v        # generate the values
    fail              # nothing else to produce
    }

  # select procedure according to first digit
  p := case s[1] of {
    "0"     :  vpred_i0
    "1"     :  vpred_i1
    default :  vpred_in
    }

  u := set()          # set for predecessors

  every t := p(s) do {  # get results from procedure
    if …                # test for valid predecessors
      then insert(u, t) # add to set
    }

  if *u = 0 then {    # no predecessors
    vpred_done[s] := nopreds
    fail
    }

  u := sort(u)        # need a list

  suspend !u          # generate predecessors

  vpred_done[s] := u  # add to memory

  fail                #  nothing else to produce

end
      …

#  initialize predecessor table for all 1– , 2–, and 3–
#  digit  numbers
procedure  vpred_init()
```

```
    local tbl

    tbl := table()

    nopreds := []           # common empty list

    # note predecessors for numbers with initial 0s
    # are needed for intermediate results
    tbl["0"] := ["0"]
    tbl["1"] := nopreds
       …
    tbl["8"] := ["4"]
    tbl["9"] := nopreds
    tbl["00"] := ["00"]
    tbl["01"] := nopreds
       …
    tbl["98"] := nopreds
    tbl["99"] := ["90"]
    tbl["000"] := ["000"]
    tbl["001"] := nopreds
       …
    tbl["998"] := nopreds
    tbl["999"] := nopreds

    return tbl

end
```

Note that providing pre-computed predecessors for all 1-, 2-, and 3-digit numbers eliminates three procedures formerly used to compute them.

Here is an example of the effectiveness of adding memory. The number tested is the 20-digit number used for examples in the last article [1].

| | |
|---|---|
| without memory: | 2760 ms. |
| with memory: | 116 ms. |

The impact of memory on procedure activity is shown in the procedure-depth histograms shown in Figures 1 and 2.

In looking at these figures, be aware that there is a great difference in the *amount* of procedure activity in the two cases. The "peaks "and "valleys" are the significant aspects.

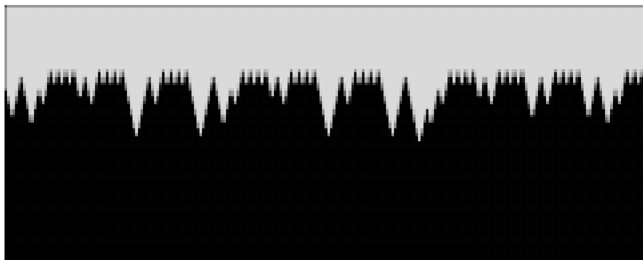


**Figure 1. Call Depth Before Adding Memory**

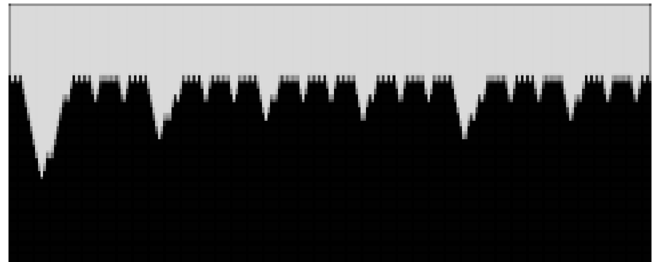


**Figure 2. Call Depth After Adding Memory**

Although providing memory for predecessors greatly increases the speed of vpred(), the time complexity is still exponential in the worst case. The exponent is smaller, but the fundamental problem remains.

## Insight

When we first started looking at the process of testing for versumness, the constraints on the last digits of a number whose initial digit is greater than 1 were evident and relatively easy to prove [3]. There was no such obvious test for numbers whose initial digit was 1, and we noted this as something to review later, but we never did came back to it.

As things stood, it was much faster, on average, to test for versumness for numbers with initial digit $i > 1$, since many can be rejected outright.

Closer inspection of versum numbers with an initial 1 also shows a pattern that can be used to reject some candidates. The pattern is not between the first and last digits but between the *second* and last digits:

| *second digit* | *possible last digits* |
|---|---|
| 0 | 0, 1, 9 |
| 1 | 0, 1 |
| 2 | 1, 2 |
| 3 | 1, 2, 3 |
| 4 | 1, 3, 4 |
| 5 | 1, 4, 5 |
| 6 | 1, 5, 6 |
| 7 | 1, 6, 7 |
| 8 | 1, 7, 8 |
| 9 | 1, 8 |

The pattern is more obvious when represented by a matrix. First digits are given in the leftmost column; possible last digits are marked in subsequent columns.

*last digit*

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | × | × | | | | | | | | × |
| 1 | × | × | | | | | | | | |
| 2 | | × | × | | | | | | | |
| 3 | | × | × | × | | | | | | |
| 4 | | × | | × | × | | | | | |
| 5 | | × | | | × | × | | | | |
| 6 | | × | | | | × | × | | | |
| 7 | | × | | | | | × | × | | |
| 8 | | × | | | | | | × | × | |
| 9 | | × | | | | | | | × | |

Note in particular that if the last digit is 1, any second digit is possible. It is, no doubt, possible to continue in this direction to find other patterns of acceptable digits, but this is enough to make a major difference in the speed with which many numbers can be tested for versumness.

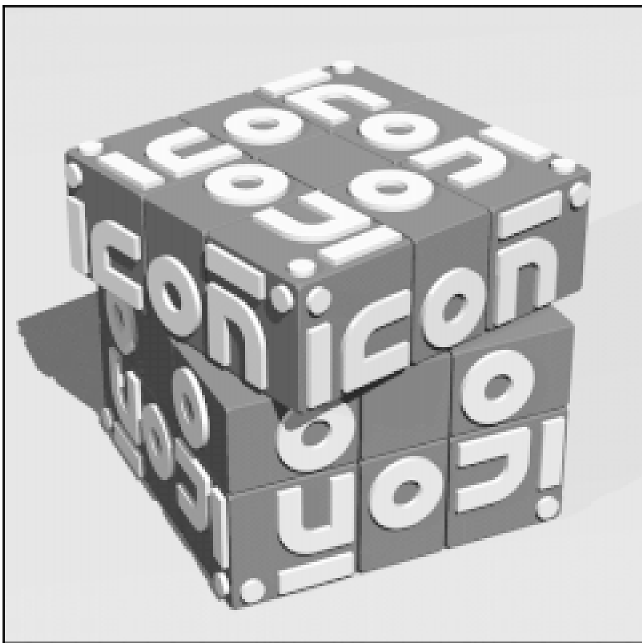
Here's one way to cast the test:

```
procedure reject(s)
   local first, second, last

   first := s[1]
   last := s[–1]

   if first > 1 then {              # "2..." ... "9..."
      if last == (first | (first – 1)) then fail
      else return
      }
   else {                           # 1 ...
      if last == "1" then fail    # "1...1" always possible
      second := s[2]
      case second of {
         "0"      :  if last == ("0" | "9") then fail else return
         "1"      :  if last == "0" then fail else return
         "2"      :  if last == "2" then fail else return
         "9"      :  if last == "8" then fail else return
         default  :  if last == (second | (second – 1))
            then fail else return
         }
      }

end
```




Note that reject() *succeeds* if the number cannot be versum.

## Loose Ends

There still are possibilities for improving the speed of versum testing. The rejection test only works at the top level, in vpred() itself. We do not completely understand why it doesn't work at lower levels (it fails only very rarely), although we have some clues. If a good rejection test could be found that worked internally in vpred_(), the speed of testing for most nonversum numbers would be improved dramatically.

We also do not completely understand the properties of intermediate results with initial zeroes. There probably is less to be gained in this area, but it's definitely a loose end.

## Conclusion

We've presented the performance of versum testing in the context of testing large numbers. Any improvements have a significant impact on small numbers also; many procedures we've used in studying aspects of versum numbers use vpred(). Cumulatively, the speed of testing is important in many areas.

## References


1. "Assault on Mount Versum", Icon Analyst 47, pp. 1-5.

2. "Versum Predecessors", Icon Analyst 37, pp. 11-15.

3. "Versum Predecessors", Icon Analyst 37, pp. 11-15.

# File-System Navigation Using VIB

The new text-list vidget [1] has considerably simplified many aspects of designing visual interfaces for Icon programs. In particular, it allows browsing through a large number of items while only using a fixed amount of screen space — a capability not available with any other vidget.

One obvious use for text-list vidgets is in navigating through a file system, moving from directory to directory, selecting files of interest, and so on.

Some operating systems provide methods for file-system navigation that applications can use. For other operating systems, an application must provide its own navigation mechanism.

Icon's dialogs provide ways for specifying the names of files for opening and saving but no way to get around in a file system or even seeing what files exist. Proving a file-navigation mechanism on a per-application basis is a formidable task. This article describes a VIB "helper" application that can be used in other VIB applications.

## The Navigator

File-system navigation is inherently system dependent. The application described here is designed for use with UNIX. Implementations for other operating systems would differ in detail but not in structure.

Figure 1 shows the navigation interface.

**Figure 1. The Navigation Interface**

A single text-list vidget displays the contents of the current directory in alphabetical order. The names of directories are followed by slashes to distinguish them from "plain" files. ../ is the directory above the current one, and ./ is the current directory, included for completeness. Clicking on the name of a directory makes it the current directory and lists its files. Clicking on the name of a plain file selects it. The two buttons at the bottom are used to dismiss the file-navigation interface. Entering a return character is equivalent to Okay, indicating acceptance of the currently selected file.

In addition to scrolling, the position in the file listing can be changed using the home and end keys, which set the position to the beginning and end of the list, respectively. Entering a character positions the listing so that the first file name beginning with the character is shown. If there is no such file name, the next in order determines the position.

Note that this navigation interface is not a complete application. It provides no way to do anything with a file or even to quit. Instead, like a dialog, it provides information for an application that uses it.

## Using the Navigator in a VIB Application

The navigator provides six global variables for communication with the application that uses it:

- nav_init, a procedure for creating the interface
- nav_window, the navigator window
- nav_root, the root vidget for the navigator
- nav_keyboard, the procedure the navigator uses for handle user keyboard input
- nav_state, the state of the navigator
- nav_file, the selected file when the navigator is dismissed.

Calling nav_init() initializes the navigator, creating its window and setting the initial values of global variables. The navigator window is hidden so that its display can be controlled by the application that uses it.

The event loop for the navigator must handle its own functionality and provide for the use of the navigator. This is a bit tricky, but it follows the general method described in the *Analyst* article on application with multiple VIB interfaces [1]. Here is a typical event loop:

```
repeat {
   case Active() of {
     &window       :  {
       root_cur := root
       keyboard_cur := shortcuts
       }
     nav_window   :  {
       root_cur := nav_root
       keyboard_cur := nav_keyboard
       }
     }
   ProcessEvent(root_cur, , keyboard_cur)
   if \nav_state then process_file()
   }
```

For each iteration of the event loop, the active window is determined to see which interface has a pending event.

If it's the application itself (&window here), the current root is set to its root (root here) and the procedure for handling keyboard events is assigned (shortcuts here).

If, on the other hand, the active window is the navigator's, the variables are set to its root and keyboard handler.

Once this is done, the event is processed. After the event is processed, the navigator's state is checked. If it is nonnull, this means that the navigator handles the event and process_file() is called to process the file.

This should appear mysterious. The navigator's window is created hidden, and hidden windows don't accept events. So where does the navigator event come from?

This is a consequence of a procedure in the application that needs a file, which might look like this:

```
procedure find_file()

   WAttrib(nav_window, "canvas=normal")

   return

end
```

All this procedure does is make the navigator interface visible so that it can accept events.

When the navigator interface is dismissed, it sets nav_state to the name of the button used to dismiss it: "Okay" or "Cancel". Since nav_state is nonnull when the navigator is dismissed, process_file() is called.

The procedure process_file() might start like this:

```
WAttrib(nav_window, "canvas=hidden")

button := nav_state
nav_state := null

if nav_state == "Cancel" then fail

input := open(nav_file) | {
  Notice("Cannot open " || image(nav_file) || ".")
  fail
  }
            …
```

First the navigation window is hidden. The variable button is used to save the navigator state before it is set to null (since the navigator interface is no longer active). Then if the navigator was dismissed with "Cancel", meaning that the user decided not to select a file, the procedure fails (it could also return; the use of fail is essentially documentation).

Note that find_file() and process_file() work cooperatively to activate and deactivate the navigator's interface.

### The Navigation Interface

Here is the code for the navigation interface. UNIX-specific code is surrounded with preprocessor conditionals to identify it and to provide a place for corresponding code for other operating systems.

#### navitrix.icn:

```
link vsetup

$include "keysyms.icn"

global directory
global dir
global file_list
global files

#  Globals used to communicate with the application
#  that uses the navigator

global nav_file
global nav_root
global nav_state
global nav_vidgets
global nav_window

procedure nav_init()
   local window_save, atts

   window_save := &window      # save current window
   &window := &null            # clear for new one
```

```
   atts := navig_atts()
   put(atts, "canvas=hidden")
   (WOpen ! atts) |
      stop("*** can't open navigation window")
   nav_vidgets := navig()          # initialize interface
   nav_window := &window          # navigation window
   &window := window_save        # restore previous
   files := nav_vidgets["files"]
   nav_root := nav_vidgets["root"]

   nav_file := nav_state := &null

   nav_refresh()

   return

end

procedure nav_files_cb(vidget, value)

   if /value then return

$ifdef _UNIX
   if value ?:= tab(upto('/')) then {
      chdir(value)
      nav_refresh()
      return
      }
$else
   Deliberate Syntax Error
$endif

   nav_file := value

   return

end

procedure nav_refresh()
   local ls, input
   static x, y

   initial {
      x := nav_vidgets["placeholder"].ax
      y := nav_vidgets["placeholder"].ay
      directory := ""
      }

$ifdef _UNIX
   input := open("pwd", "p")
$else
   Deliberate Syntax Error
$endif

   WAttrib( nav_window, "drawop=reverse")
   DrawString(nav_window, x, y, directory)
   DrawString(nav_window, x, y, directory := !input)
   WAttrib(nav_window, "drawop=copy")

   close(input)

   file_list := [ ]

$ifdef _UNIX
   ls := open("ls –a –p .", "p")
$else
   Deliberate Syntax Error
$endif

   every put(file_list, !ls)

   VSetItems(files, file_list)

   close(ls)

   return

end

procedure nav_okay_cb()

   if /nav_file then {
      Notice("No file selected.")
      fail
      }

   nav_state := "Okay"

   return

end

procedure nav_locate(e)
   local i
   static pos

   initial pos := list(1)

   every i := 1 to *file_list do {
      if file_list[i] >>= e then break
      }

   pos[1] := i

   VSetState(files, pos)
```

## Supplementary Material

Supplementary material for this issue of the *Analyst*, including images and code, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia48/ia48sub.htm

```
    return
  end

  procedure nav_shortcuts(e)

  case e of {
    "\r"          :  nav_okay_cb()
    Key_Home  :  VSetState(files, 1)
    Key_End    :  VSetState(files, *file_list)
    default      :  if type(e) == "string" then
      nav_locate(e)
      }

    return

  end

  procedure nav_cancel_cb()

    nav_state := "Cancel"

    return

  end

  #===<<vib:begin>>===
  procedure navig_atts()
    return ["size=294,412", "bg=pale gray",
     "label=Navitrix"]
  end

  procedure navig(win, cbk)
  return vsetup(win, cbk,
    ["navig:Sizer:::0,0,294,412:Navitrix",],
    ["cancel:Button:regular::86,378,49,20:Cancel",
      nav_cancel_cb],
    ["files:List:w::13,50,273,314:",nav_files_cb],
    ["okay:Button:regular::21,378,49,20:Okay",
      nav_okay_cb],
    ["placeholder:Button:regularno::20,22,65,17:    ",],
    ["refresh:Button:regular::224,378,56,20:refresh",
      nav_refresh],
    ["border:Rect:grooved::18,374,55,28:",
      nav_okay_cb],
    )
  end
  #===<<vib:end>>===
```

## A Browser Using the Navigation Application

Here is a simple application that illustrates the use of the navigation interface. As written, it provides only for invoking the navigation interface (find from the File menu or @F as a keyboard shortcut).

If a file is selected, it is displayed in a "read-only" text list, where it can be examined. Figure 2 shows the browser after a file has been selected.

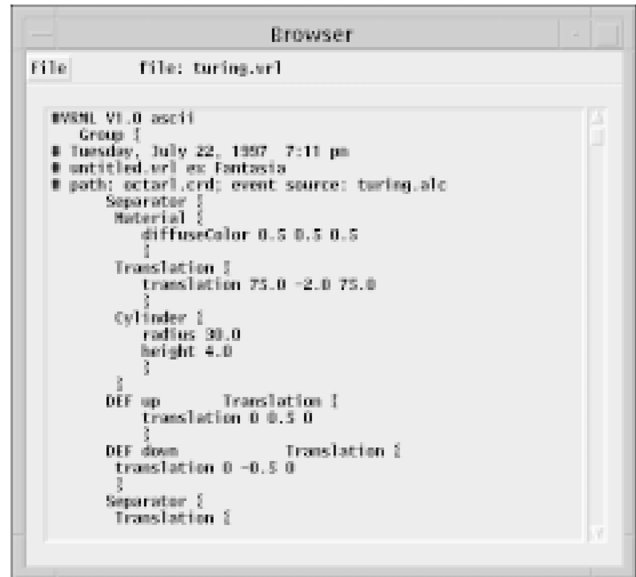



**Figure 2. A File Browser**

This application is only designed to show how the navigation interface might be used. It could, of course, be made more capable, such as providing the ability to rename and delete files.

**browser.icn:**

```
link navitrix
link vsetup

global placeholder
global vidgets

$define LineLength  75
$define FileLength 500

procedure main()
  local root, root_cur, keyboard_cur

  nav_init()

  vidgets := ui()

  root := vidgets["root"]
  placeholder := vidgets["placeholder"]

  repeat {
    case Active() of {
      &window      :  {
        root_cur := root
        keyboard_cur := shortcuts
        }
      nav_window   :  {
        root_cur := nav_root
        keyboard_cur := nav_keyboard
        }
```

```
    }
  ProcessEvent(root_cur, , keyboard_cur)
  if \nav_state then process_file()
  }

end

procedure process_file()
  local input, file_list, button
  static list_vidget, x, y, name_old

  initial {
    list_vidget := vidgets["list"]
    x := placeholder.ax + TextWidth("file: ")
    y := placeholder.ay + WAttrib("leading") – 2

    name_old := ""
    }

  WAttrib(nav_window, "canvas=hidden")

  button := nav_state
  nav_state := &null

  if button == "Cancel" then fail

  input := open(nav_file) | {
    Notice("Cannot open " || image(nav_file) || ".")
    fail
    }

  file_list := [ ]

  every put(file_list,
    left(entab(!input), LineLength)) \ FileLength

  VSetItems(list_vidget, file_list)

  close(input)

  WAttrib("drawop=reverse")
  DrawString(x, y, name_old)
  DrawString(x, y, \nav_file | "")
  WAttrib("drawop=copy")

  name_old := \nav_file

  return

end

procedure file_cb(vidget, value)

  case value[1] of {
    "find  @F"   :  find_file()
    "quit  @Q"   :  exit()
    }

  return

end

procedure find_file()

  WAttrib(nav_window, "canvas=normal")

  return

end

procedure shortcuts(e)

  if &meta then case map(e) of {
    "f"    :  find_file()
    "q"    :  exit()
    }

  return

end

#===<<vib:begin>>===
procedure ui_atts()
  return ["size=526,402", "bg=pale gray",
    "label=Browser"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
  [":Sizer:::0,0,468,402:Browser",],
  ["file:Menu:pull::0,3,36,21:File",file_cb,
    ["find  @F","quit  @Q"]],
  ["list:List:r::14,44,500,340:",],
  ["menubar:Line:::0,26,525,26:",],
  ["placeholder:Label:::88,7,42,13:file: ",],
  )
end
#===<<vib:end>>===
```

## Acknowledgment

In the article on the text-list vidget [1], we neglected to acknowledge that the original version was implemented by Jason Peacock under Clint Jeffery's direction.

Without his work, Icon's interface toolkit would not have this valuable vidget.

## Reference

1. "Multiple VIB Interfaces", Icon Analyst 42, pp. 1-4.

## Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/

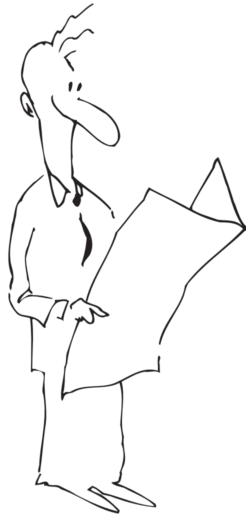
## Programming Tips

### Avoiding Unnecessary Storage Allocation

An expression such as

```
["red", "blue", "green"]
```

sometimes is called a "literal list". This gives the mistaken impression that the list is a constant that is evaluated when the program is compiled. Instead, such an expression creates a new list whenever it is encountered during program execution.



Although it is possible to detect lists with constant elements at compilation time, creating such lists at that time and putting references to them in the code could have disastrous effects. For example, in

```
primaries := ["red", "blue", "green"]
            …
put(primaries, "cyan", "magenta", "yellow")
```

the second expression would modify the contents of the "constant" list. If the first expression were evaluated a second time, it would have six elements, not the apparent three.

A new kind of list whose length could not be changed has been suggested, but that would add all kinds of complexity to the language and its implementation.

When it is known that a list will not be changed after its creation, repeated re-creation can be avoided by the simple expedient of assigning it to a static variable in the initial clause for the procedure:

```
procedure blend(c)
   static primaries

   initial primaries := ["red", "blue", "green"]
            …
```

instead of

```
procedure blend(c)
   local primaries

   primaries := ["red", "blue", "green"]
            …
```

If such a list is shared by several procedures, a global variable can be used instead.

It is, of course, good practice to use static or global variables for constant values and initialize them before they are used. But it's also extra work and often ignored or overlooked.

It's also easy to overlook situations in which the same list is created over and over but never changed. For examples lists are used for arguments in some dialogs. An example is

```
TextDialog(
   ["Your chance to", "find a treasure!"],
   ["jewel", "coin"],
   ["emerald", "gold eagle"],
   [10, 12]
   )
```

which produces a dialog that looks like this:

In this example, four "constant" lists are created every time the dialog is used. This unnecessary list creation can be avoided as follows:

```
      …
static captions, type, choices, widths

initial {
   captions := ["Your chance to", "find a treasure!"]
   type := ["jewel", "coin"]
   choices := ["emerald", "gold eagle"],
   widths := [10, 12]
   }
      …
TextDialog(captions, type, choices, widths)
```

Of course, other structure-creation operations have the same properties as list creation, but other kinds of structures are less often used as "constants".

Don't expect to improve program performance much using this technique. Structure creation itself is a fast process. The more serious

concern about any kind of unnecessary storage allocation is that it may impact program performance by increasing the amount of time spent in garbage collection, a relatively slow operation.

In fact, this is an example of what we call "micro improvements" in program performance. If, however, you can't think of anything better to do, such even small improvements may be worthwhile. There's a less obvious aspect of making such improvements in a program: In the process, you may notice other things about the program that can lead to more significant improvements.

We plan to have an article in a future issue of the Analyst about the relative importance of various approaches to improving program performance.


The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
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
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–project@cs.arizona.edu



THE UNIVERSITY OF
ARIZONA®
TUCSON ARIZONA

*and*

*Bright Forest Publishers*
*Tucson Arizona*



© 1998 by Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend

All rights reserved.


## Subscription Renewal

For many of you, this is the last issue in your present subscription to the Analyst. If so, you'll find a renewal form in the center of this issue. Renew now so that you won't miss an issue.

Your prompt renewal helps us by reducing the number of follow-up notices we have to send. Knowing where we stand on subscriptions also lets us plan our budget for the next fiscal year.

## What's Coming Up

In the next issue of the Analyst, we plan to have another article in the series on character patterns. This article will focus on an application to help locate and encode patterns. We'll be back with yet another article on versum numbers, this time on the distances between successive ones — a topic that motivated much of our work on character patterns.

We have a backlog of articles for the **Graphics Corner**; we'll try to squeeze one of these in the next issue.