The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1990
Number 3

In this issue …

Program Readability

Everyone has somewhat different personal preferences for laying out the text of a program. While it's a little extra work to make a program attractive and readable, it's usually worth it: The ease with which a program can be understood depends to a considerable extent on how it's laid out. And even if you don't expect someone else to read your programs, you'll probably have to look at one of your own programs long after you wrote it. If you've ever had the experience of looking at one of your old programs and saying to yourself "what in the world is this all about?", you know what we mean.

Of course, readability has a lot to do with programming techniques. It's often possible to do the same thing several ways in Icon and some of Icon's features may lead you to write "cute" but impenetrable code where something simple, straightforward, and understandable will do just as well. The guidelines that follow do not attempt to address this issue; they're just some simple rules for program layout. You may not agree with all these guidelines, but give them some thought.

1. Put global, record, and link declarations at the beginning of your program where they are easy to find.

2. Put the main procedure before all other procedures so it's easy to see where your program starts.

3. Arrange other procedures in some logical way, either by category or alphabetically by procedure name.

4. Use white space liberally to reduce the "density" of your code and to set apart logically connected parts of your programs. Use blank lines to set off logically connected segments of code.

5. Use blanks around binary operators, except in subscripting expressions, to set operators apart from their operands.

6. Avoid the use of semicolons to separate expressions. Instead, put the expressions on separate lines.

7. Adopt a "paragraphing" style for laying out nested expressions and use it religiously. Indent in a consistent manner. There are many paragraphing styles (the second edition of the Icon language book illustrates one of these). The choice is more a matter of taste than correctness. The rule is consistency.

8. Use braces to clearly delineate the extent of complicated expressions, even when braces are not required syntactically.

9. Use parentheses to show the intended grouping of operators and operands in complicated expressions, even if parentheses are not necessary.

10. Avoid long lines. Instead, continue expressions on as many lines as are needed for easy readability. See the first two issues of The Icon Analyst for the proper techniques for breaking lines. Similarly, continue long string and cset literals on several lines if that's necessary to make them readable. See Appendix A of the second edition of *The Icon Programming Language* for the method of continuing quoted literals.

11. Use literal escape sequences for "special characters", even if these characters can be entered directly in a program. For example, use "\t" instead of just typing a tab. Do not use the character's internal code (such as char(9)).

12. Comments are all-important in program readability. Don't procrastinate in providing them, but remember that you may change your program after you write your comments. Go back over your comments when your program is "finished".

13. Favor narrative blocks of comments prior to the code to which they refer over short, fragmentary comments on individual lines of code.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)
(128.196.128.118 or 192.12.69.1)

Writing Portable Icon Programs

Most of the Icon programs you write probably are for your own use and are never read by anyone else, much less used by others on other computer systems.

If you're writing Icon programs for use by others, however, such as submissions to the Icon program library, you should give some attention to making these programs as portable as possible, so that they can be run on a wide variety of computers and operating systems.

Icon itself is basically quite portable. All implementations of Icon to date, whether for personal computers or mainframes, are based on the same source code. Consequently, unlike most programming languages, you won't find different dialects or implementation quirks that make moving a program from one platform to another a tedious and unpleasant task. Of course, Icon has evolved over a period of time and you can't expect a program written to run under Version 8 of Icon to run under earlier versions. We're assuming Version 8 here.

Although Icon programs generally are quite portable, there are system-specific extensions in some implementations of Icon and some underlying differences in systems that cause portability problems.

Portability is a matter of degree. A program may be truly portable and run on any system without modification. Another program may, say, run on 90% of all systems, while yet another may require changes to run on any system other than the one on which it was written. Some programs are truly non-portable and only run on one system.

Most of the issues that need to be considered when trying to write portable Icon programs fall into three classes:

- language features to avoid
- character-set differences
- operating system differences

Language Features

Language features to avoid when writing portable programs fall into two classes:

- basic language features that are not implemented for all systems
- implementation-specific extensions

The Icon keyword &features, which generates all the major characteristics of the system on which it is evaluated, provides a place to start. The first value generated by &features is the operating system (for example, MS–DOS). The second value generated is the character set (ASCII or EBCDIC). The remaining values list the supported features.

Of these features, two are basic in the sense that they are part of the Icon programming language proper, while the remaining features are more system-specific. The two basic features that may not be supported on a specific implementation of Icon are co-expressions and large-integer arithmetic.

Co-expressions require assembly-language code to switch from one evaluation context to another. This code may be easy, difficult, or even (rarely) impossible to write, depending on the architecture of the computer in question. Even if the code is easy to write, it may not have been done yet (many implementations of Icon for specific computers have been done by individuals around the world). At the present time, all the implementations of Icon that are distributed by the Icon Project support co-expressions, with the exception of some of the many variants of the UNIX implementation. As of this writing, there are 56 such variants, of which 16 do not support co-expressions. In a few cases co-expressions are supported but do not always work correctly. Even in cases where co-expressions are supported and work correctly, there may be problems because of the large amount of memory that co-expressions require. A program that uses co-expressions intensively may work on one system but not on another.

If you want your programs to be truly portable, do not use co-expressions.

Large-integer arithmetic is written entirely in C and can be provided, in principle, for all implementations of Icon. Icon, however, is a memory-intensive program and it barely fits in the limited memory of some personal computers. Large-integer arithmetic adds a lot of code bulk to Icon. For this reason, large-integer arithmetic is not supported on some personal computer implementations of Icon. Specifically, it's not supported on the Amiga, the Atari ST, and some versions of the MS-DOS implementation of Icon.

Again, avoid large-integer arithmetic if you want your programs to be portable.

There are other language features that are not part of the basic Icon language, but which are available in specific implementations. For example, ProIcon has extensive capabilities for manipulating windows and menus. MS-DOS implementations of Icon contain a set of functions specifically designed for the MS-DOS environment. Some UNIX implementations provide a way of saving the execution state of an Icon program so that it can be restarted later. Obviously, programs that use such features are not portable.

Other non-basic features are more general in nature:

- environment variables
- pipes
- keyboard functions
- shell commands

Environment variables (called different things on different systems) allow communication of named values between the operating system and applications. Environment variables are supported by most implementations of Icon. The exceptions are ProIcon for the Macintosh (the Macintosh operating system has no concept of environment variables) and the Atari ST, when run in some environments.

Since most Icon programs do not use environment variables, their support usually is not a portability issue. It's worth noting that getenv(s), which returns the value of the environment variable s if it is set, fails if environment variables are not supported (on the principle that in the absence of environment variables, none can be set).

Pipes, which allow the output of one process to be fed to the input of another, require (at least) a multi-tasking operating system. Pipes are supported only in the UNIX, VMS, and OS/2 implementations of Icon. An attempt to open a file as a pipe, which is how Icon uses the facility, causes a run-time error on implementations of Icon that do not support pipes. While pipes allow many things to be done from within an Icon program that could not be done otherwise, they should not be used in a program that's intended to be portable.

The keyboard functions getch(), getche(), and kbhit() allow direct input to an Icon program from the console of the computer on which it's running. These keyboard functions often are used in interactive applications, but they are supported only on "personal computers", not UNIX, VMS, or IBM 370 systems.

Programs that use keyboard functions are apt to be nonportable anyway, since they typically depend on the characteristics of specific devices, such as monitors.

The function system(s) issues s as a command to be executed by the operating system ("shell"). This only makes sense if the operating system supports a "command-line interface" (the Macintosh operating system does not). The use of system() is problematical in any event, since it may not work properly even on implementations that support it in principle. For example, MS-DOS implementations of Icon support system(), but the command may fail for lack of memory. Furthermore, shell commands vary from system to system, and there's little in the way of commands that is portable.

Character Sets

Most implementations of Icon use the ASCII character set. The exceptions are the CMS and MVS operating systems for the IBM 370 architecture, which use the EBCDIC character set.

Character set differences can cause all kinds of problems in program portability. Some of the problems are subtle and difficult to understand.

Both ASCII and EBCDIC have 256 different characters, represented internally by numerical codes from 0 to 255. The problem is that the two character sets associate "printable" characters and control functions with numerical codes in different ways.

In ASCII, for example, the numerical code for "A" is 65, while in EBCDIC, it is 193. (The Icon function ord(s) returns the code for the one-character string s.)

Not only is the association of codes for printable characters different in ASCII and EBCDIC, but the codes are in fundamentally different relative positions for important classes of characters. Uppercase letters have smaller codes than lowercase ones in ASCII, while the opposite is true in EBCDIC. Similarly, the digits have smaller codes than the letters in ASCII, while the opposite is true in EBCDIC.

These differences in "collating sequences" show up in lexical comparison and (hence) in sorting strings.

The confusion is compounded by different ways of handling EBCDIC internal character codes (inside an Icon program) and external codes (in files) in EBCDIC implementations of Icon. In the 370 implementations of Icon distributed by the Icon project, the internal and external codes are the same. For example, an "A" is 193 in both. This uniformity of codes means that an Icon program that sorts strings on a 370 produces what 370 users expect in their environment. But the results may be different from those if the Icon program is run on an ASCII system. On the other hand, the 370 VM/CMS implementation of Icon done by Walter Schiller in Germany translates external EBCDIC codes to internal ASCII ones on input and vice versa on output. This implementation produces the same results when sorting strings on ASCII and EBCDIC systems, but produces results in EBCDIC environments that are different from what EBCDIC users may expect. The character set shown by &features is ASCII for this implementation, even though it runs on EBCDIC systems.

The 370 implementations of Icon that produce the expected results on EBCDIC systems introduce portability problems between ASCII and EBCDIC systems. On the other hand, it's not entirely clear what portability means in this context — expected results for the local environment or identical results in different environments.

The problems with character sets are not limited to lexical comparison and sorting. The value of &ascii is essentially meaningless when EBCDIC internal codes are used, although the 370 implementations distributed by the Icon Project approximate as best as possible the expected ASCII characters. It's certainly a good idea to avoid the use of &ascii in programs that are intended to be portable.

Other portability problems can arise from the way "special" characters are represented in programs. For example, the value of char(9) is a tab character in ASCII but not in EBCDIC. Any reference to a character by its code or position in the character set should be avoided. Instead, literal escape sequences, such as "\t", should be used when writing portable programs. Icon handles these in the expected way, regardless of the character set.

Also do not assume that any particular numerical code (such as one above 127) is safe to use as a character that will not appear in "ordinary text". In ASCII, some systems, such as the Macintosh, use nearly every character for text. In EBCDIC, the printable characters are distributed from the lower numerical codes through the higher ones.

It's also worth noting that IBM 370 systems do not have the same set of control characters as ASCII systems. The characters produced by Icon's control escape sequences, such as "\^C" may mean nothing or something different from what they do on ASCII systems.

Another matter related to character sets concerns the characters that are "printable" in EBCDIC. While all printable characters are associated with numerical codes in a standard way in ASCII, this is not true of EBCDIC. In fact, there are several different commonly used EBCDIC mappings between codes and printable characters. These mappings all agree on the codes for letters, numbers, and "common" punctuation marks, but they disagree on characters like braces and brackets. The associations between numerical codes and EBCDIC printable characters used in the 370 implementations of Icon distributed by the Icon Project are listed in Appendix B of the second edition of *The Icon Programming Language*.

Worse yet, most 370 terminals do not provide a way of entering braces and some can't handle brackets either. (Line printers without lowercase letters also are common in 370 environments, but that's a different problem.)

Since braces and brackets are an important part of the syntax of Icon programs, Version 8 of Icon supports the following "digraph" equivalents:

standard character	digraph
{	$(
}	$)
[	$<
]	$>

These digraphs work on ASCII implementations of Icon also. In principle, if you're trying to write a program to be portable to an EBCDIC system, you should use these digraphs. That's a bit much to ask, however. Fortunately, the Icon program library for EBCDIC systems has a program for converting from standard characters to digraphs, and conversely. Since the problem with braces and brackets is peculiar to the EBCDIC world, it seems reasonable for 370 Icon programmers to perform the necessary conversions.

Operating System Differences

Most Icon programs do not deal directly with the operating system on which they run. Icon itself has most of the necessary facilities for operations like opening and closing files, as well as renaming and deleting them. Nonetheless, these facilities must communicate with the operating system.

Some operating systems (notably ones for the IBM 370) don't support hierarchical file systems (directories), but have fundamentally different ways of organizing collections of data. Other operating systems have different methods for specifying directory paths and logical devices. All of these concepts are fundamentally non-portable.

File naming conventions differ from operating system to operating system. As far as portability is concerned, it's a good idea to avoid creating files or referring to specific files if you don't have to. Standard input and standard output are the best choices in situations where they will do. If your program must create or reference file names of its own choosing, be careful and be conservative. Keep file names short and simple. Stick to letters (even the underscore cannot be used in file names on some systems). Realize that upper- and lowercase letters are distinct on some systems and equivalent on others. If you use a period to separate part of a file name from its "extension", use the MS-DOS convention: at most 8 characters before the period and at most 3 after. Never use more than one period in a file name.

Some systems, such as MS-DOS, allow specific devices (like a printer) to be opened like files. This kind of facility is obviously non-portable.

There are subtler problems with input and output. Some systems, noticeably those that run on the IBM 370, have strict limits on the lengths of lines. An attempt to write a line longer than the limit results it its being broken up into multiple lines.

Output to a monitor is always problematical. You'd probably expect long lines to be treated differently by different devices (wrapped or truncated), but you might not be prepared for MPW on the Macintosh, in which a line written by writes() is overwritten from the beginning by the next line of output. There's no hope of writing portable programs to work with monitors, and you shouldn't try.

Uncertainties

You can count on at least 32-bit integer arithmetic in Icon, even in the absence of support for integers of arbitrarily large magnitude. A few implementations of Icon, such as the one for the Cray-2, have 64-bit integers. This possible difference is unlikely to affect most programs.

Icon's random number sequence, which is used in all random selection operations, is the same on most systems, but you can't count on it.

Sorting in Icon uses the C run-time library routine qsort(). Some implementations of qsort() are stable but others are not (in an unstable sort, equivalent values may change in relative position). Consequently, sorting values other than numbers and strings may produce slightly different results on different systems.

Summary

While there are many considerations in writing programs to be portable over a wide range of systems, the main ones are:

- Do not use co-expressions, large-integer arithmetic, or any system-specific functions.
- Be careful about character set differences; do not refer to characters by their numerical codes.
- Avoid the use of named files and file structures.
- Keep input and output simple and modest with respect to line length.

String Scanning

String scanning probably causes more frustration than any other major feature of Icon. Most of the problems with string scanning concern how to use it — how to formulate scanning expressions to solve specific string analysis and synthesis problems. Other problems with string scanning relate to style and efficiency.

In order to use string scanning effectively, it's important to understand its essential characteristics. Icon programmers often think of string scanning as a complex feature. In fact, string scanning is very simple. The power of string scanning comes from other aspects of Icon: its notion of strings, generators and goal-directed evaluation, and its repertoire of string-analysis functions. This is the first of a series of articles that describe various aspects of string scanning and its use. Much of the material in this first article will be familiar to practiced Icon programmers, but the viewpoint may be new.

Conceptual Basis

As described in the second edition of the Icon language book, the design of string scanning is based on the observation that most kinds of string analysis can be cast in terms of a succession of computations on a single string. This string, called the *subject*, is the focus of attention during string scanning. Different string scanning expressions may, of course, have different subjects. But during the evaluation of any one scanning expression, the subject (normally) does not change.

Another observation is that most string-analysis operations can be cast in terms of the examination of the subject at some specific location. The examination may cause the location to be changed — hence the term *scanning*. Most often, the numerical value associated with the location is not of interest. The interesting thing usually is what character is there or what substring starts or ends there. In analyzing a sentence, for example, certain words and their relative positions may be important, but their specific locations in terms of character count in the sentence usually are not.

Scanning Environments

The subject, together with the current location, called the *position*, constitute a *scanning environment* associated with a scanning expression. We'll denote the scanning environment by {*subject*, *position*}. This is not an Icon value; it's just a notation for talking about scanning environments. As mentioned above, the subject portion of a scanning environment normally does not change during the evaluation of a scanning expression, but the position usually does.

A change in the scanning environment as the result of changing the position normally is implicit and happens as a side effect of an analysis operation. String-analysis operations usually can be written without any explicit reference to the subject or the position. This eliminates clerical detail and tedious (and error-prone) numerical computations that are characteristic of lower-level forms of string analysis.

Program execution begins with the scanning environment {"", 1}, as if scanning were about to take place at the beginning of an empty string. New scanning environments are created by string scanning expressions, which have the form

expr1 ? *expr2*

where *expr1* is called the *subject expression* and *expr2* is called (somewhat misleadingly) the *analysis expression*. In most cases, the subject expression is simple, such as a string-valued variable. But, as is generally true in Icon, the subject expression can be any expression. The result of evaluating the subject expression must, of course, be a string or a type that can be converted to a string. We'll describe some possibilities for more complicated kinds of subject expressions in a subsequent article.

Before the subject expression is evaluated, the current scanning environment is saved. The evaluation of the subject expression establishes a new scanning environment with the its value as the subject and the initial position at the beginning of this subject. For example, if the value of word is "exemplary", the initial scanning environment for

```
word ? while write(move(1))
```

is {"exemplary", 1}.

When a scanning expression is finished, it restores the scanning environment that it saved before it began.

Matching Functions

The functions move(i) and tab(i), which should be familiar to all Icon programmers, illustrate the concepts of scanning and the way the position is changed as a side effect of their evaluation. These functions are called *matching functions*, since they return the substring of the subject between the positions before and after their evaluation — the portion of the subject *matched*. Matching functions fail if the new position would be out of the bounds of the subject. Thus, failure of a matching function provides a natural way to control loops in string scanning.

The function move(i) increments the position by i. Since move(i) causes relative movement in the subject, its argument usually is some specific number, as in the example at the end of the preceding section. When using move(i) to scan, the actual value of the position is not evident — it just moves along the subject. In the example above, move(1) changes the scanning environment successively to {"exemplary", 2}, {"exemplary", 3}, …, and finally to {"exemplary", 10}, while writing the characters matched.

On the other hand, tab(i) sets the position to a specific value. Except in the case of scanning strings with data in fixed-position fields, the argument of tab(i) usually is not a specific value. Instead, it often is supplied by a string analysis function, as in

```
sentence ? while tab(upto(&letters)) do
   write(tab(many(&letters)))
```

Again, it's not necessary to know the specific positions.

A common use of tab(i) is to set the position at the right end of the subject by using tab(0). Note that although the argument here is an integer, the specific position is not specified or even of interest.

An important aspect of matching functions is data backtracking. When a matching function produces a result, it suspends, even though it cannot produce another result (there is only one way to set the position to a specific value). If a subsequent expression fails, the suspended matching function is resumed, and it restores the position to the value it had prior to the evaluation of the matching function, thus restoring the scanning environment to its former value.

Data backtracking performed by matching functions assures that alternative expressions start at the same place. For example, in

```
sentence ? {
   (tab(5) & find("the")) | tab(0)
   }
```

if tab(5) succeeds but find("the") fails, the resumption of tab(5) restores the position to its previous value (at the beginning of the subject) and tab(0) starts at the same position as tab(5) did.

Maintenance of Scanning Environments

Scanning expressions are on a par with all other expressions in Icon. Consequently, scanning expressions can occur in conjunction, as in

(*expr1* ? *expr2*) & (*expr3* ? *expr4*)

Scanning expressions also can be nested, as in

(*expr1* ? (*expr2* ? *expr3*))

Procedures called in analysis expressions also can contain scanning expressions. For example, in

expr1 ? p()

p() itself may contain scanning expressions. This situation amounts to dynamic nesting, and it occurs more frequently in practice than the static form of nesting shown above.

In order for such constructions to behave in a reasonable way, Icon maintains multiple scanning environments. Scanning environments are global with respect to procedure calls, but they are local to scanning expressions.

As mentioned earlier, execution begins with the scanning environment: {"",1} (an empty, zero-length subject). When a scanning expression is evaluated, it saves the current scanning environment and creates a new one. If the scanning expression fails, it restores the previously saved scanning environment. If the scanning expression suspends, its scanning environment is saved (since it may be resumed) and the previous scanning environment is restored. A scanning environment remains in existence until its corresponding analysis expression fails or until it is no longer possible to resume it. This occurs as the result of control structures that discard suspended generators.

Because of the possibility of scanning expressions in conjunction as well as nested scanning expressions, the structure connecting saved scanning environments is a tree, not a stack. (This is true of suspended generators, also.)

In general, the tree of scanning environments is rooted in the scanning environment associated with the initiation of program execution as described above. There are two ways that the tree of scanning environments can grow. One is horizontally, as in expressions such as

(*expr1* ? *expr2*) & (*expr3* ? *expr4*) & …

The other is vertically, as in expressions such as

(*expr1* ? (*expr2* ? (*expr3* ? (*expr4* ...))))

In horizontal growth of the scanning environment tree, an analysis expression is suspended during the evaluation of a subsequent expression. In vertical growth, before an analysis expression completes evaluation, a scanning expression that is nested within it is evaluated. Vertical growth usually appears in programs in the form of matching procedures that themselves contain scanning expressions, as mentioned above.

As an example, consider the evaluation of the following expression:

```
("abc" ? move(2 | 1)) &
("defg" ? (tab(4) ? move(1 | 2)))
```

Assuming that there is no other surrounding expression, the evaluation of

```
"abc" ? move(2 | 1)
```

causes the scanning environment tree to become

$$\begin{tikzcd}
\{\text{""}, 1\} \arrow[d] \\
\{\text{"abc"}, 3\}
\end{tikzcd}$$

When

```
"abc" ? tab(2 | 1)
```

suspends,

```
"defg" ? (tab(4) ? move(1 | 2))
```

is evaluated. The tree of scanning environments grows horizontally. After the evaluation of tab(4), the tree is:

$$\begin{tikzcd}
\text{\{"", 1\}} \arrow[d] \arrow[drr] & & \\
\text{\{"abc", 3\}} & & \text{\{"defg", 4\}}
\end{tikzcd}$$

Evaluation of the nested scanning expression then causes the tree of scanning environments to grow vertically:

$$\begin{tikzcd}
\text{\{"", 1\}} \arrow[d] \arrow[drr] & & \\
\text{\{"abc", 3\}} & & \text{\{"defg", 4\}} \arrow[d] \\
 & & \text{\{"def", 2\}}
\end{tikzcd}$$

If the expression above appears in a context that causes it to be resumed, as in

```
(("abc" ? move(2 | 1)) &
("defg" ? (tab(4) ? move(1 | 2)))) & expr
```

where *expr* fails, then the expression move(1 | 2) is resumed and the last scanning environment is changed:

$$\begin{tikzcd}
\text{\{"", 1\}} \arrow[d] \arrow[drr] & & \\
\text{\{"abc", 3\}} & & \text{\{"defg", 4\}} \arrow[d] \\
 & & \text{\{"def", 3\}}
\end{tikzcd}$$

Further resumption produces no new result for this expression, resumption of tab(4) produces no new result, the second scanning expression in the mutual evaluation produces no new result, and move(2 | 1) in the first scanning expression in the mutual evaluation is resumed. At this point, the scanning environment tree again has the form

$$\begin{tikzcd}
\text{\{"", 1\}} \arrow[d] \\
\text{\{"abc", 3\}}
\end{tikzcd}$$

Note that two scanning environments were discarded as the result of the failure of scanning expressions.

The second result for move(2 | 1) changes this environment to

$$\begin{tikzcd}
\text{\{"", 1\}} \arrow[d] \\
\text{\{"abc", 2\}}
\end{tikzcd}$$

At this point the second scanning expression in the mutual evaluation is evaluated again, and the tree of scanning environments grows again in a fashion similar to that illustrated above. The tree of scanning environments reverts to a single root node only when all alternatives in the mutual evaluation have been produced.

Note that all the nodes along the right edge of the tree of scanning environments correspond to expressions whose evaluation is incomplete and are “active”, while all other nodes correspond to inactive expressions that may produce another result if they are resumed because of failure of expressions corresponding to nodes to their right.

Next Time

This article mainly provides food for thought and background. Subsequent articles will be more concerned with actually using string scanning than with the conceptual aspects discussed here.

The material here is nonetheless important to using string scanning effectively. Look at some of the scanning expressions you’ve written and see how it applies to them.

Generators

In the last issue of the Analyst, we discussed the basic aspects of expression evaluation in Icon: success, failure, and generation.

Generation lies at the heart of Icon. Generation arises naturally from the concept of computations that may have more than one alternative. The alternation control structure,

expr1 | *expr2*

allows you to formulate this kind of computation explicitly, in terms of possible alternatives. And, of course, if there are several alternatives, these can be given in a compound alternation:

expr1 | *expr2* | *expr3* | … | *exprn*

Iteration

As mentioned in the previous article on expression evaluation, a generator only produces an alternative if it is resumed: There must be some context that needs an alternative for a generator to produce more than one result. This context may be implicit, as in

```
counter = (1 | 2 | 3 | 4)
```

in which the alternation expressions are resumed to produce alternatives only if the numerical comparison operation fails. For example, if the value of counter is 2, only two of the four possible alternatives are produced.

Sometimes it is useful to force a generator to produce all its alternatives. This is expressed in Icon explicitly by the *iteration* control structure:

every *expr1* do *expr2*

which repeatedly resumes *expr1* and evaluates *expr2* for each alternative produced by *expr1*. And example of iteration is

```
every i := (1 | 2 | 3 | 4) do
   if counter > i then write(i)
```

which assigns the values 1, 2, 3, 4 to i and compares each with counter, writing only the ones that are less than counter.

Built-In Generators

Some kinds of generators are used so frequently that Icon provides them as part of its built-in computational repertoire. For example, integers often are needed in sequence. The expression

```
i to j by k
```

produces the integers from i to j in increments of k. If the by clause is omitted, the increment defaults to 1. Thus, the previous example could be written as

```
every i := 1 to 4 do
   if counter > i then write(i)
```

This form is easier to read than the explicit alternation. It also is more concise, especially for large ranges. More important, the range for the sequence need not be known when a program is written, while it must be fixed and known in advance if explicit alternation is used.

You may have noticed the similarity between the every-do loops shown above and for loops that are commonly found in other programming languages. Consider

```
every i := 1 to 100 do
   write(sqrt(i))
```

Although this resembles a for loop, it really is composed of iteration and a generator. There are many other possible combinations of iteration with generation that provide more flexibility and conciseness than are possible with for loops.

There's also another possibility. The argument of a function can be a generator, as in

```
write(sqrt(1 to 100))
```

Standing alone, this expression just writes 1. But put it in an iteration expression such as

```
every write(sqrt(1 to 100))
```

and you have a compact expression for writing the square roots of the integers from 1 to 100.

Two things here deserve note. The do clause in iteration is optional; if there's nothing to do, you can leave it out. The other important matter is that a generator that's inside another expression is resumed if an alternative for the expression is needed. In the example above, iteration requests the next alternative. Although write() and sqrt() themselves have no alternatives, there are alternative arguments for sqrt() and hence for write().

As you probably know, there are other built-in generators — but perhaps fewer than you might expect. In fact, only five functions and two operators are generators:

find()	locations of substrings
upto()	locations of characters
bal()	locations of characters following balanced strings
i to j by k	integers in finite sequence
seq()	integers in endless sequence
!x	elements of structures, etc.
key()	keys in table

There also are four keywords that are generators:

&features	implementation features
&collections	garbage collections
&regions	storage region sizes
&storage	storage region utilizations

We'll explain later why there are so comparatively few built-in generators in Icon's repertoire.

Another Look at Alternation

Our examples of alternation so far all have had simple arguments. Suppose an argument of alternation is a generator, as in

```
(1 to 10) | (21 to 30)
```

What results does this expression generate? There are complicated, step-by-step ways of describing alternation, but there's also a very simple answer: The alternation of two expressions produces the results of the first expression followed by the results of the second expression. Thus,

```
every write((1 to 10) | (21 to 30))
```

writes 1, 2, … 10, 21, 22, … 30.

You may find it helpful to remember this simple characterization of alternation, since you may find times when you want to write a generator that produces two sequences of results, one after the other.

Programmer-Defined Generators

Although Icon has relatively few built-in generators, you can make your own generators by using procedures. The idea is simple: Instead of just returning a result from a procedure, you can write the procedure to produce a result, but suspend, so that it can be resumed to produce alternatives. All you need to do to accomplish this is to use suspend instead of return when you want a procedure to produce a result.

Consider, for example, a procedure to generate the square roots of the integers between i and j, inclusive:

```
procedure roots(i,j)
  every k := i to j do
    suspend sqrt(k)
end
```

The procedure computes a square root and produces this result by suspending, instead of returning. If the procedure call is resumed, evaluation continues in the procedure immediately after the suspend expression, namely at the end of the do clause. The every loop continues, the counter is incremented, and another result is produced by suspension, and so on. If the procedure is resumed repeatedly until the end of the integer sequence and hence the every loop, control flows off the end of the procedure body. That is, this resumption of the suspending procedure fails and produces no alternative.

There's an important feature of suspend that allows procedures such as this one to be written more compactly. The suspend expression resumes its argument for alternatives in the manner of the every expression. In the procedure above, the argument of suspend is not a generator, so this feature is not used. But since suspend resumes its argument, it's not necessary to have both every and suspend. The following, simpler form will do:

```
procedure roots(i,j)
  suspend sqrt(i to j)
end
```

When a suspended call of this procedure is resumed, the generator that is the argument of sqrt() is resumed and the procedure suspends again with the new result.

There's no limit to what you can do with programmer-defined generators. In fact, since generation is such an important aspect of expression evaluation in Icon, when you're deciding how to formulate a computation using a procedure, you should ask yourself if the computation might be better cast as a generator than, say, as a procedure that just returns a single value and has to be called many times. The answer to such a question depends on the nature of the computation, the context in which it is used, and whether generation is a natural approach. Think about Icon's built-in generators in this regard — why the computations they perform are better cast in terms of generation than the production of single values.

Limitation

The function seq(), mentioned earlier, generates an unending sequence of integers: 1, 2, 3, … . The idea that a generator may produce an infinite number of alternatives may be alarming. What's to prevent it from going into a loop, out of control?

In the first place, a generator does not produce its alternatives spontaneously. There must be some context that causes a generator to be resumed in order for it to produce an alternative. For example,

```
i := seq()
```

just assigns 1 to i and then goes on to the next expression.

Nonetheless, infinite generators *can* get out of control:

```
every write(sqrt(seq()))
```

writes the square roots of the positive integers endlessly; there's nothing to stop iteration from repeatedly resuming seq().

Icon provides a way to prevent such problems. The *limitation* control structure

expr \ i

limits *expr* to at most i results. Consequently,

```
every write(sqrt(seq() \ 1000))
```

writes only the square roots of the integers from 1 through 1,000.

The limit need not be applied directly to the generator; it can be moved out to apply to the entire argument of every:

```
every write(sqrt(seq())) \ 1000
```

In the example given here, it would be more natural to use a finite generator that did not need to be limited:

```
every write(sqrt(1 to 1000))
```

In most cases, however, there is not a convenient finite form for an infinite generator.

Repeated Alternation

Just as loops are useful in conventional computation for performing computational tasks repeatedly, there are times when you'll find it useful to produce alternatives repeatedly. If the number of repetitions you want is fixed, you can write an explicit compound alternation:

expr1 | *expr2* | … | *exprn*

But if you want this to go on indefinitely, you can't use such an expression. Icon provides *repeated alternation* for this purpose:

|*expr*

This expression generates the alternatives for *expr*, generates them again, and so on, endlessly. Repeated alternation, with an exception we'll mention soon, is an infinite generator. Consider a simple case:

```
|1
```

This expression generates 1, 1, 1, … endlessly. If the argument of repeated alternation is a generator, its alternatives are produced repeatedly:

```
|(1 to 3)
```

generates 1, 2, 3, 1, 2, 3, 1, 2, 3,… .

As with any generator, repeated alternation can be limited:

```
|(1 to 3) \ 5
```

generates 1, 2, 3, 1, 2. Note that limitation applies to the total number of alternatives, not the number of repetitions.

The important exception to the infinite generation of repeated alternation is that it stops if its argument ever fails to produce a single result. This may seem a bit strange. If an expression fails, why put it in repeated alternation?

One of the reasons for this special termination is to prevent a disaster if the argument of repeated alternation fails. Consider

```
|(1 = 0)
```

If repeated alternation kept going, expression evaluation would get stuck; since (1 = 0) never produces a result, the expression above would never suspend so that any further evaluation could take place.

Of course, you wouldn't write an expression like this intentionally, but there are plenty of less obvious ways to get into this potential bind.

Aside from this concern, there are computationally useful aspects for the termination condition for repeated alternation. Some expressions succeed at one time and fail at another. An example is read(), which produces a result as long as there is data left in the input file, but fails if there is not. Thus,

```
|read()
```

is an expression that generates the lines from the input file but stops when there are no more. If you like to program in terms of generators, you can write

```
every process(|read())
```

instead of the more straightforward

```
while process(read())
```

We recommend the second form, but the first one is worth thinking about.

The Rationale for Generators

There's an interesting question here. Why *isn't* read() a generator instead of just a function that produces a line every time it is evaluated? In fact, if generators are so great, why are there so few in Icon's built-in repertoire? Why isn't everything a generator?

It isn't hard to answer the last question — generators are only reasonable for computations in which there are natural alternatives. It makes no sense for addition to be a generator.

But isn't it reasonable to think of reading lines as producing alternatives and hence cast the operation as a generator? In a sense, this view *is* reasonable. The reason why such operations are not generators lies in the possibility of undesired resumption. Suppose, for example, that Icon's random-number operation were a generator. Then in an expression like

```
if i > ?j then …
```

if i were less than the value produced by ?j, ?j would be resumed to produce an alternative. This would continue forever if i were greater than j. In any event, there would be no way to know how many alternative random numbers were produced, even if the comparison eventually succeeded.

Put another way, generators are powerful, but they also are dangerous. For this reason, Icon's built-in repertoire of generators is limited to operations that are most naturally cast in terms of alternative results of computation.

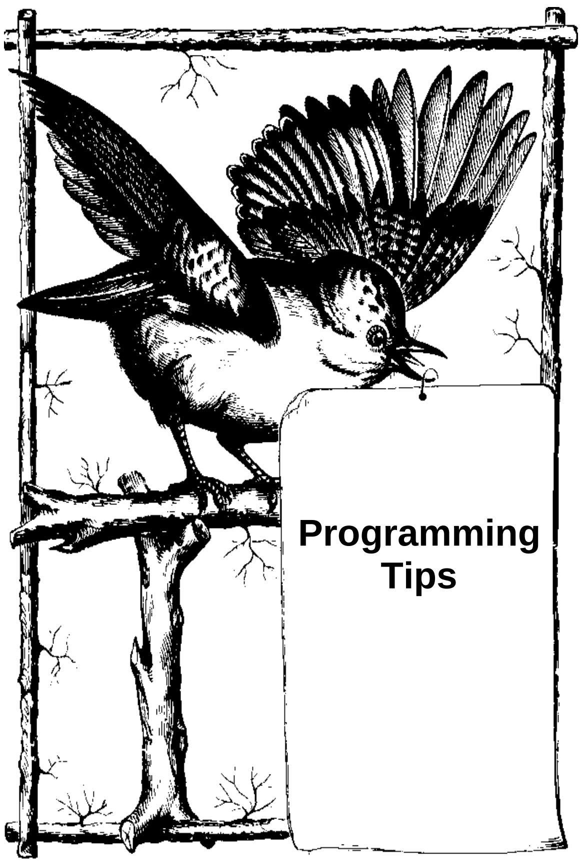
If you're not happy with this, you can write procedures that serve as generators. Or you can always turn a non-generator into a generator:

```
|?i
```

is a generator of random numbers.

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 copy for airmail postage to other countries.



Case Expressions

Sometimes it's helpful to remember that a case selection expression need not be a constant. In fact, it can be a generator. If it is, the selector matches if any generated value matches. The usefulness of this programming technique is illustrated by the following example:

```
case response of {
   "y" | "Y"    :   action := "yes"
   "n" | "N"    :   action := "no"
   }
```

Think of the kinds of expressions that are generators and you should see lots of other interesting possibilities for this technique.

It may also be helpful to know that a case expression in Icon is equivalent to if … then … else if …, in which the control expressions use general value comparison. For example, the case expression above is equivalent to

```
if response === ("y" | "Y") then action := "yes"
else if response === ("n" | "N") then action := "no"
```

Automatic Type Conversion

Lots of things in a programming language go toward making it easy or hard to write programs. Automatic type conversion is one of the features that makes it easy to write programs in Icon. You can, for example, write an integer without having to explicitly convert it to a string, as in

```
every write(2 ^ (0 to limit))
```

Automatic type checking and conversion are integral parts of Icon and you should use them without reservation or suspicion. Mistrust in automatic type conversion sometimes shows up in the form of unnecessary explicit conversions, as in

```
every write(string(2 ^ (0 to limit)))
```

with a comment like "just to be sure". Such explicit type conversions are not only unnecessary, but they are actually inefficient, not to mention confusing to someone who reads them.

Despite the usefulness of automatic type conversion, it sometimes hides inappropriate data and inefficiency. We frequently see expressions like this:

```
upto(" ")
```

The function upto() expects a cset. If it's given a string, as in this case, the string is automatically converted to a cset and the expressions works just as if it had been written

```
upto(' ')
```

Although the first form works properly, the type conversion takes time. The amount of time required for the conversion is proportional to the length of the string being converted.

If you're just starting to learn Icon, having to remember what functions expect csets is just one more chore. And it's easy for even an experienced Icon programmer to use double quotes where single quotes are appropriate. It's worth looking over your programs and seeing if the types of literals you've used are appropriate.

From the Wizards

The following little program by Anthony Hewitt was mentioned in *Icon Newsletter 32*. We're including it here, since it deserves some discussion. You may or may not think it constitutes wizardry. Granted, it's a bit tricky, but then wizards often are. Wizardry or not, it's clever, concise, and much in the spirit of idiomatic Icon. Here's the program:

```
procedure main()
   write(s := !&input)
   every write(s ~==:= !&input)
end
```

It has a simple function — to filter out adjacent duplicate lines in standard input.

You might pause at

```
s := !&input
```

It's equivalent to

```
s := read()
```

and just reads the first line of standard input. The form used here is becuase of the next line, which is the key to the program:

```
every write(s ~==:= !&input)
```

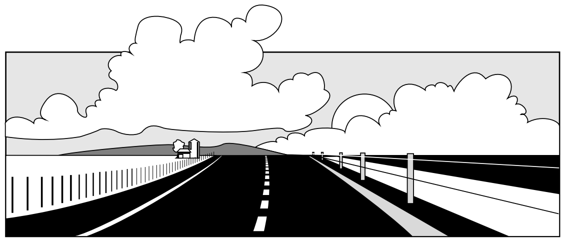
This expression reads a line of input. If it's different from the previous line, which is the value of s, it assigns the new line to s and writes it. If the new line is the same as the previous one, the augmented assignment fails and nothing is written. The every control structure keeps this going until the file is exhausted.

If you prefer read() to !&input, you need to construct a generator to keep reading. Repeated alternation does this, and the result is

```
procedure main()
   write(s := read())
   every write(s ~==:= |read())
end
```

The two lines in the procedure body can even be combined into one:

```
procedure main()
   every write((s := read()) | (s ~==:= |read()))
end
```



What's Coming Up

In the next issue of the Analyst, we'll continue the series on string scanning with an article on how to formulate scanning expressions and give some examples.

There also will be articles on programs that write programs, large-integer arithmetic, and memory utilization.

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

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

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

(602) 621-8448

FAX: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

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

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona