The Icon Analyst

In-Depth Coverage of the Icon Programming Language

August 1991
Number 7

In this issue ...

String Synthesis

One criticism of string scanning in Icon is that it is heavily oriented toward analysis and provides little help in synthesis. In fact, all of Icon's string synthesis facilities are relatively low-level. Imagine a somewhat different feature that might replace string scanning: *string synthesis* that produces an object string while analyzing a subject string.

The framework used for string scanning can be extended to string synthesis by adding two variables to the environment to replace the scanning environment by a *synthesis environment* containing a subject, an object, a subject position, and an object position: {*subject*, *object*, *sposition*, *oposition*}.

The idea is that the analytic portion of string scanning works in string synthesis as before, but an object string is synthesized and the object position specifies the current position of interest in the object string.

As with any proposed new language feature, the details are important, even crucial. It seems reasonable to formulate synthesis facilities that are analogous to the analysis facilities, but it's not so clear just how synthesis should be cast conceptually.

Here's a proposal for a set of synthesis functions. The idea is to modify the model of string scanning given in the last issue of the Analyst to model a string-synthesis facility.

For a string synthesis expression

expr1 ? *expr2*

the expression *expr1* provides the subject as in string scanning, and the object is initially the empty string. The positions in both initially are 1. As in string scanning, *expr2* is then evaluated. While it can do anything, it generally analyzes the subject and synthesizes the object. The result of the string synthesis expression (if it succeeds) is the value of the object (not the result of *expr2*).

Here's the challenge: Model string synthesis with the mapping

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

and provide appropriate procedures for Bform() and Eform(). Use the global variables subject, s_pos, object, and o_pos for the state variables. (Do not use &subject and &pos.)

In addition, provide the following analysis and synthesis procedures:

- smove(i) and stab(i), which are the same as move(i) and tab(i) in string scanning.
- omove(i) and otab(i), which are the same as smove(i) and stab(i), except they apply to the object, not the subject.
- oplace(s), which inserts s in the object following the current position in it, changes the position to the end of the inserted string, and returns the new value of the position.
- xswap(), which swaps the values of the subject and the object, and sets both positions to 1.
- odelete(i), which deletes the i characters of the object following the current position. It does not change the position, but returns it as value.
- spos(i), which is the same as pos(i) for string scanning.
- opos(i), which is the same as spos(i), but for the object, instead of the subject.

Here's an example of the use of these procedures:

```
write(
  "abcde" ? {
     while smove(1) do
       oplace(smove(1))
     oplace("]")
     otab(1)
     oplace("[")
     }
  )
```

The value written is [bd].

Take this as an exercise and see what you can do with it. Don't hesitate to try to improve on the procedures suggested here.

We'll give our solution to this exercise in the next issue of the Analyst.

Variant Translators

You may have come across situations in which you'd like to transform an Icon program into something slightly different. There were two examples of this in Issue 6 of the Analyst: "Modeling String Scanning" and "Evaluation Sandwiches". In modeling string scanning, scanning expressions need to be transformed into nested procedure calls:

expr1 ? *expr2* → Escan(Bscan(*expr1*),*expr2*)

In evaluation sandwiches, all expressions need to be transformed:

expr → 2(before(),*expr*,after())

Another example similar to modeling string scanning occurs in "String Synthesis" in the preceding article in this issue of the Analyst.

The usual way to handle transformations like these is to write a preprocessor that performs the desired transformations. Writing a preprocessor is a pattern-matching and replacement problem. Since Icon is particularly good for these kinds of operations, writing such a preprocessor sounds easy. But it really isn't.

In order to preprocess a program, it's necessary to parse most of the language in which the program is written. Icon has a rather complex syntax.

Even in the case of something as simple as transforming string scanning operations, the parsing is not that easy. It isn't just a matter of finding question marks. The subject and analysis expressions must be identified. Both can be arbitrarily complicated and may extend over many lines. String scanning can be nested, which leads to recursive processing. Quoted literals and comments may contain text that looks like string scanning. A program being preprocessed may contain syntactic errors, and so on. As you get into it, the task of writing a preprocessor becomes more and more complicated.

You may be willing to make compromises, like ignoring the possibility that text in quoted literals and comments may resemble scanning operations. You may not care if your preprocessor malfunctions if the input is syntactically erroneous. Such compromises, however, lead to *ad hoc* preprocessors that are not robust, complete, or correct. It's also likely to be a lot more work than you thought when you started, even with compromises.

If you have the necessary knowledge and skills, you could go inside Icon's translator and change it to generate the kind of code you want. While this approach, done carefully, can produce a robust and demonstrably correct variant of Icon's translator, it also requires a lot of work — so much so that you're not likely to actually do it. You're certainly not going to use this approach frequently or just in order to try out something like modeling string scanning.

We had concerns like this when we were developing Icon. We had many ideas for experimental facilities that we wanted to try. Most of these facilities could be cast as transformations of "standard" Icon to some "extended" Icon, perhaps with procedures to model new features, much as in modeling string scanning. In other words, we needed preprocessors, but not at the expense of a great deal of time and effort.

This problem led us to develop a system for producing correct, complete, and robust preprocessors with only a small percentage of the effort required for conventional approaches. Such preprocessors are called *variant translators*, since they involve variations on Icon's standard translator.

Translation

Icon's translator, known behind the scenes as itran, converts Icon programs into *ucode*, which is an assembly language for an imaginary computer. Ucode files are eventually converted to binary *icode*, which is an "executable" image of the Icon program. Thus, itran translates from one language, Icon, to another, ucode. An example of this translation is shown in the box below. It's not necessary to understand ucode for the purposes of this article; it's enough to know that it's a representation of Icon operations in another language.

```
line ? {  # write each one          line    1
   while write(move(1))      →      mark    L1
   }                                var     0
                                    line    2
                                    bscan
                               lab  L2
                                    mark0
                                    var     1
                                    var     2
                                    int     0
                                    line    3
                                    invoke  1
                                    invoke  1
                                    unmark
                                    goto    L2
                                    line    2
                                    escan
                                    unmark
                               lab  L1
```

Icon to Ucode Translation

In order for itran to translate Icon programs into ucode, it must, of course, understand Icon's syntax. itran contains the usual lexical analyzer for identifying tokens like literals and operators, and it has a parser for processing the higher levels of Icon's syntax. The parser produces "actions" that build trees that subsequently are traversed in order to generate ucode.

The key to a variant translator, which we'll call vitran, is to change the code generator to produce Icon code instead

of ucode. With this change, the simplest variant translator essentially does nothing. It reads in a program and writes it out. The only differences between the input and output are in the arrangement of the text, the insertion of semicolons implied by line breaks in the input, and the removal of comments in the input. See the box below.

```
line ? {     # write each one        line ? {
  while write(move(1))          →    while write(move(1))
  }                                  };
```

Identity Translation

That is, the output is an Icon program that is semantically equivalent to the input program. Such a translator is called an *identity translator*.

Specifying Variants

There isn't much use for an identity translator; it just illustrates the idea. The important next step is to modify the code generator to produce a desired transformation. Nothing else needs to be changed to do this — the lexical analyzer and parser work as before. As you might guess, modifying the code generator to transform string scanning expressions into nested procedure calls doesn't require much effort — especially since the output language is Icon, not ucode. See the box below.

```
line ? {     # write each one
  while write(move(1))          →    Escan(Bscan(line),{
  }                                  while write(move(1))
                                     });
```

Variant Translation

Since vitran is an Icon-to-Icon translator (preprocessor), its output can now be compiled like any other program. In the case of something like modeling string scanning, the necessary procedures can be linked together with the main program.

The basic idea here is a very powerful one; with a few extensions, it can handle changes in the input syntax as well as changes in the generated code. Variant translators have been used for many research projects ranging from list scanning [1] to the experimental programming languages Rebus [2] and Seque [3]. The interesting thing about Rebus is that its syntax looks a lot like Icon's, but the generated code is SNOBOL4 [4]!

To build variant translators of any complexity, software support is needed. Even transforming all expressions into evaluation sandwiches requires extensive modifications of the code generator, albeit routine ones. Icon's variant translator system does just that. Variants can be given in a high-level specification language and most of the steps in actually building a variant translator are automated. It is not, for example, necessary to get into Icon's code generator at all.

Specifications

The specification of a variant translator consists of definitions for Icon's syntactic types followed by the desired translations. A syntactic type is given in a functional (macro) form followed by the translation as a concatenation of tokens. For example, the standard (identity) translation for

return *expr*

is specified as

```
Return(x,y)   "return "   y
```

The first argument, x, corresponds to the reserved word return, while the second argument, y, corresponds to the argument of the return expression. Thus, the identity specification shown here simply causes return expressions to be translated as-is. A variant specification, such as

```
Return(x,y)   "{suspend "   y   "; fail}"
```

would cause all return expressions to be transformed into suspend and fail expressions (a transformation of dubious merit, but you might imagine putting something useful between them).

The variant translator system comes equipped with a complete set of built-in identity specifications. Only variations from these need to be specified — variants override the corresponding identity specifications. For example, to produce a variant translator that transforms string scanning expressions into nested procedures requires only one variant specification:

```
Bques(x,y,z)   "Escan(Bscan("   x   "),"   z   ")"
```

The name Bques identifies the binary question-mark operator. (Another variant specification really should be included to account for augmented string scanning; it's correspondingly simple.)

The specification system also provides shorthand notations for syntactic classes, such as binary (infix) operators. Consequently, only one variant specification is needed to handle all binary operators in the same way.

Other Considerations

Some potentially interesting transformations require making minor modifications to Icon's syntax. For example,

expr1 ?? *expr2*

was used for modeling the list-scanning facility mentioned earlier, using procedures analogous to those for modeling string scanning. Since ?? is not an Icon operator, it has to be added for the variant translator. Adding an operator requires minor modifications to Icon's lexical analyzer and parser.

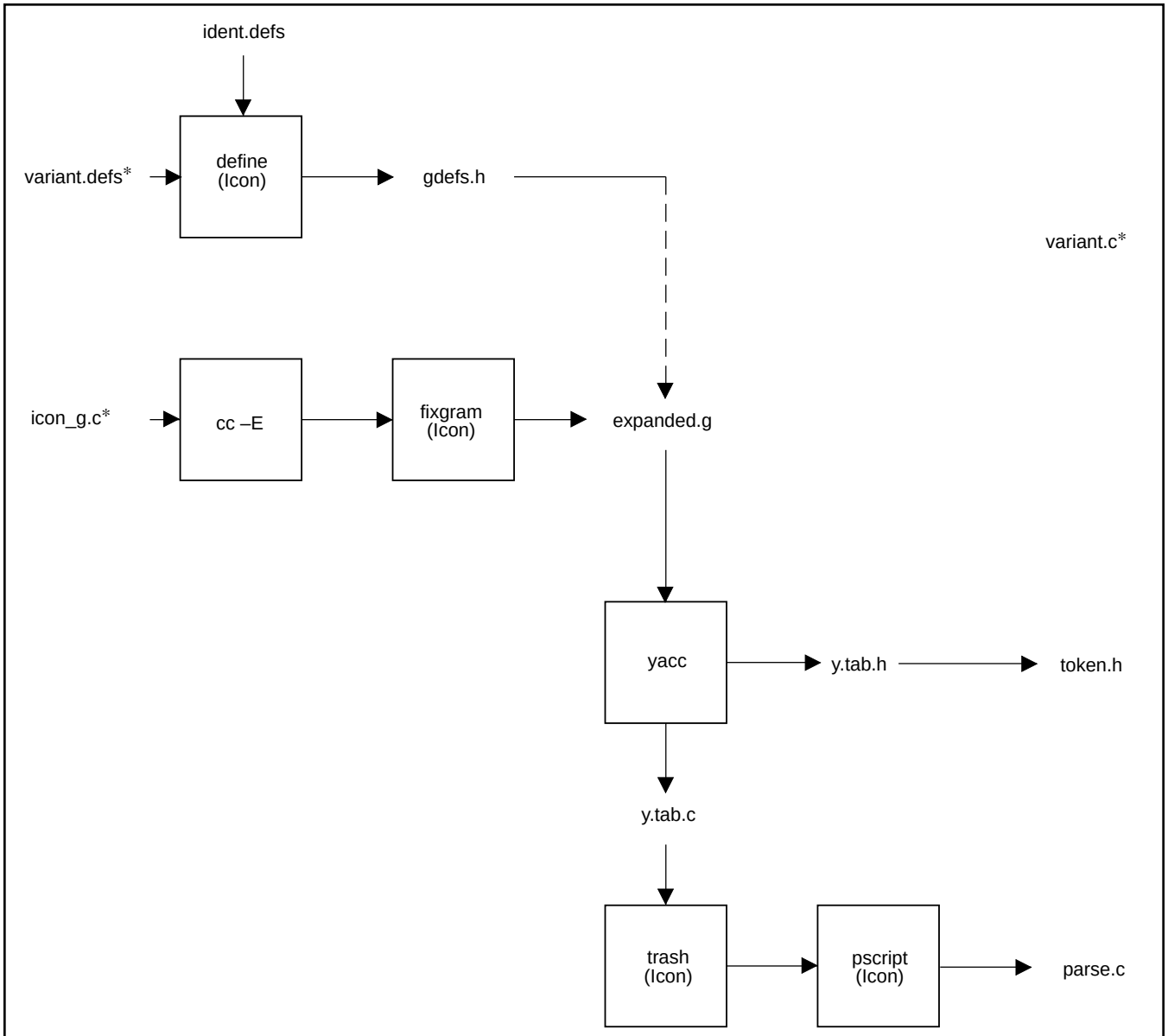
Fortunately, most of Icon's lexical analyzer is built from high-level specifications and Icon's parser is produced by Yacc [5]. Consequently, simple changes like adding an operator do not require a lot of low-level coding.

Building a Variant Translator

Building a variant translator requires considerable computational resources. It can't be done on all personal computers, but it's routine on UNIX systems and it can be handled on several other systems.

To begin with, the variant translator system requires a production-quality C compiler, Icon, and Yacc. It also requires a lot of memory, since Icon's Yacc grammar is large and uses macros extensively.

The schematic diagram at the bottom of this page shows the main components of Icon's variant translator system. Files listed on the left are processed as shown to produce the files on the right. The files on the right, together with other files for Icon's translator, are then compiled and linked to produce the final result. Only the most important files are shown in the diagram. Files flagged with an asterisk contain variant specifications. The file variant.defs contains any variant translation specifications. It's almost always needed. The specifications in variant.defs override the standard ones in ident.defs. The result of processing these files is a set of macro definitions in gdefs.h that are included by expanded.g (hence the dashed line). The file icon_g.c contains a stylized form of the Yacc grammar for Icon. After macro preprocessing, it provides the input to Yacc. After some postprocessing, the C files needed are ready for inclusion in the building of vitran. One other file, variant.c, is needed. Ordinarily it is empty, but if additional C functions or declarations are needed for the translation, this is where they go. The Icon programs fixgram, trash, and pscript are used for various minor changes that are needed in file formats.



Building a Variant Translator

If you have access to Icon on a UNIX system, it's easy to build variant translators — almost as easy as pushing a button. For other systems, it depends on what resources you have and in particular on whether you have a robust version of Yacc.

It's worth noting, however, that a variant translator can be built on one system for use on another. Thus, variant.c, token.h, and parse.c can be built on a UNIX system and compiled on an MS-DOS system. See Reference 6 for general information about building variant translators.

Further Thoughts

There are advantages and disadvantages to variant translators. They are fast, faithful to the syntax of Icon, robust, and (assuming they are specified properly) correct. Variant translators are not, however, particularly portable or easy to build except under UNIX.

The idea behind variant translators is not limited to the present C- and Yacc-based system. If anyone ever wrote a complete, correct, robust, and well-structured identity translator for Icon in Icon, it should be a fairly easy matter to build a specification system on top of it and have most of the advantages of the present variant translator system without the need for C and Yacc. A variant translator produced by such an Icon-based system would not run as fast as one produced by the present system, but it would be much more portable, as would an Icon-based variant translator system itself.

Don't feel obligated to go out and write an identity translator for Icon in Icon — but if you feel so inclined, give some thought to what would be needed to adapt it to variants.

References

1. *Unifying List and String Scanning in Icon*, Allan J. Anderson and Ralph E. Griswold, Technical Report TR 83-4, Department of Computer Science, The University of Arizona, 1983.

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

3. "Seque: A Programming Language for Manipulating Sequences", Ralph E. Griswold and Janalee O'Bagy, *Computer Languages*, Vol. 13, No. 1 (1988), pp. 13-22.

4. *The SNOBOL4 Programming Language*, second edition, Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1971.

5. *Yacc — Yet Another Compiler-Compiler*, S. C. Johnson, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, 1978.

6. V*ariant Translators for Version 8 of Icon*, Ralph E. Griswold and Kenneth Walker, Technical Report TR 90-4, Department of Computer Science, The University of Arizona, 1990.

Result Sequences

An important aspect of "thinking in Icon" is understanding generators and their role in programming so thoroughly that you can use them as problem-solving tools.

Most descriptions of generators treat them in a dynamic fashion, emphasizing where and when they produce their results. An alternative view of generators is more abstract and deals with the sequence of results they are capable of producing in an abstract way, as mathematical objects. While this abstract view of *result sequences* is somewhat removed from the actual process of programming, it can provide insights to the use of generators and how to think about them.

In the dynamic view of generators, the focus is on what happens as a generator is evaluated, suspends with a result, and is resumed — the events during expression evaluation and how a generator behaves with respect to these events. In the abstract view of result sequences, the focus is on the results a generator is *capable* of producing. A result sequence can be viewed as the results a generator would produce if resumed repeatedly. For example,

every *expr*

forces *expr* to produce all its results. Taken together, these results comprise the result sequence for *expr*.

Notation

To describe and manipulate result sequences as abstract objects, some notation is needed. A result sequence is enclosed in braces (which have nothing to do with braces in Icon programs) to emphasize its status as an abstract object. For example, the result sequence for 1 to 5 is $\{1, 2, 3, 4, 5\}$ and the result sequence for seq() is $\{1, 2, 3, \ldots\}$, where the ellipses indicate an unending (infinite) sequence. Note that there is no problem with the abstract concept of an infinite result sequence.

Other useful notation is:

S(*expr*)	the result sequence for *expr*
L(*expr*)	the length of the result sequence for *expr*
Φ	the empty result sequence, { }
$S_1 \oplus S_2$	the concatenation of result sequences S_1 and S_2
S^i	The result sequence S concatenated with itself i times

Some examples using this notation are:

L(1 to 5) = 5

S(1 to 5) $\oplus$ S(1 to 5) = {1, 2, 3, 4, 5, 1, 2, 3, 4, 5}

S(&fail) = Φ

Control Structures

The usefulness of dealing with result sequences is illustrated by the alternation control structure

expr1 | *expr2*

A description of alternation in dynamic terms, "if *expr1* is resumed but produces no result, then *expr2* is evaluated …" is convoluted and obscures the fundamental simplicity of alternation. In terms of result sequences,

$$S(expr1 \mid expr2) = S(expr1) \oplus S(expr2)$$

That is, the result sequence for the alternation of two expressions is the concatenation of their result sequences. Put another way, the results produced by the alternation of two expressions consist of the results of the first expression followed by the results of the second expression.

Another illustration of the conciseness of result sequence notation in describing expression evaluation is the result sequence for a compound expression:

$$S(\{expr1; expr2; \ldots exprn\}) = S(exprn)$$

This identity points out two things: (1) the results produced by a compound expression depend only on the last expression and (2) a compound expression can fail or be a generator — something that you may not have thought about.

Failure drives Icon control structures. The special role of failure can be described in terms of the empty result sequence, Φ, or in terms of the lengths of result sequences:

$$S(\text{if } expr1 \text{ then } expr2 \text{ else } expr3) = \begin{cases} S(expr2) & \text{if } S(expr1) \neq \Phi \\ S(expr3) & \text{if } S(expr1) = \Phi \end{cases}$$

or

$$S(\text{if } expr1 \text{ then } expr2 \text{ else } expr3) = \begin{cases} S(expr2) & \text{if } L(expr1) > 0 \\ S(expr3) & \text{if } L(expr1) = 0 \end{cases}$$

Similarly, the special termination condition for repeated alternation can be expressed as:

$$S(|expr) = \begin{cases} S(expr)^{\infty} & \text{if } L(expr) > 0 \\ \Phi & \text{if } L(expr) = 0 \end{cases}$$

Result sequences also provide a compact way of describing the general properties of classes of expressions. For example, the looping control structures while, until, and every all have empty result sequences, as in

$$S(\text{while } expr1 \text{ do } expr2) = \Phi$$

In dynamic terms, these control structures fail when they terminate. This actually only is true if a loop terminates normally (for example, when *expr1* fails in the example above). If a loop terminates because of break, its result sequence is quite different, since

$$S(\ldots \text{ break } expr \ldots) = S(expr)$$

where this means the result sequence of the control structure in which the break is evaluated. Thus, in the case of termination of a loop as a result of break, the result sequence for the loop can even be a generator!

Operations and Goal-Directed Evaluation

Another useful pedagogical aspect of result sequences occurs in expressions in combination with goal-directed evaluation. For this topic, it is helpful to classify functions and operations (we'll use operations here, since the distinction is only syntactic) in terms of the characteristics of their result sequences. For example, given a binary operation ⊗ and specific values *v1* and *v2*, what is the nature of the result sequence for *v2* ⊗ *v2*?

Most operations in Icon are *monogenic*. That is, they produce exactly one result for specific arguments. For example,

$$S(1 + 2) = \{3\}$$

and, in general

$$S(v1 + v2) = \{v1 + v2\}$$

Thus, for a monogenic operation,

$$L(v1 \otimes v2) = 1$$

Other operations are *conditional* and produce one result or none, depending on the specific arguments. For example,

$$S(2 > 1) = \{1\}$$
$$S(1 > 2) = \{\} = \Phi$$

Thus, for a conditional operation,

$$L(v1 \otimes v2) \leq 1$$

For generative operations, there is no simple relation; in most cases the minimum number of results is 0 and the maximum depends on the specific values.

The examples of operations given above are for specific arguments. If the arguments are produced by expressions, the result sequences are more complicated, reflecting the fact that the expressions may generate many values, so that the operation is performed on many combinations of arguments. For monogenic operations, there is a simple relationship between the lengths of the result sequences. For example,

$$L(expr1 + expr2) = L(expr1) \times L(expr2)$$

This relationship reflects the "cross-product" form of evaluation, in which generators are resumed in a last-in, first-out

fashion. See [1] and [2] for a more detailed treatment of this matter.

For conditionals, the product is an upper bound, as in:

$$L(expr1 \ > \ expr2) \le L(expr1) \times L(expr2)$$

There is no upper bound that applies to all generators.

One interesting monogenic operation is conjunction, *expr1 & expr2*. Consider first specific argument values *v1* and *v2*:

$$S(v1 \ \& \ v2) = \{v2\}$$

For a general expression as the second argument,

$$S(v1 \ \& \ expr) = S(expr)$$

This is, the value of the first argument has no affect on the result sequence. Now suppose the first argument is some expression. In this general case,

$$S(expr1 \ \& \ expr2) = S(expr2)^{L(expr1)}$$

That is, the result sequence is the result sequence for *expr2* concatenated with itself for each value in the result sequence for *expr1*. This may seem a bit strange, but consider a simple example:

$$S((1 \text{ to } 3) \ \& \ 5)) = \{5, 5, 5\}$$

If you prefer something more concrete, try this in a program:

```
every write((1 to 3) & 5)
```

The Distributivity of Alternation

The power of result sequences for describing the characteristics of expression evaluation in Icon is shown by the following two identities relating to the distributivity of alternation:

$$S(v(expr1) \mid v(expr2) \mid \ldots \mid v(exprn)) = S(v(expr1 \mid expr2 \mid \ldots \mid exprn))$$

$$S(v1(expr) \mid v2(expr) \mid \ldots \mid vn(expr)) = S((v1 \mid v2 \mid \ldots \mid vn)(expr))$$

Concrete examples of these identities are:

$$S(\text{find(s1)} \mid \text{find(s2)}) = S(\text{find(s1} \mid \text{s2))}$$

and

$$S(\text{integer(x)} \mid \text{string(x)}) = S((\text{integer} \mid \text{string})(\text{x}))$$

If you are interested in how such identities can be proved, see References 1 and 2, which describe an elementary calculus of result sequences.

Using Result Sequences

By now, this may seem very abstract and esoteric, especially in light of the introductory remarks about "thinking in Icon". There are several ways, however, in which result sequences can be used to formulate expressions.

Result sequences can provide a method of formulating expressions that produce desired results. For example, to generate the lowercase letters five times, you might be inclined to write

```
!&lcase | !&lcase | !&lcase | !&lcase | !&lcase
```

But recalling the result sequence for conjunction, you can write a more concise (and more easily generalized) expression:

```
(1 to 5) & !&lcase
```

Similarly, the first identity in the preceding section concerning the distributivity of alternation allows you to write

```
tab(10) | tab(5) | tab(2) | tab(1)
```

as

```
tab(10 | 5 | 2 | 1)
```

The second identity in the previous section suggests a way of formulating expressions that is less obvious— the expression that is applied to a list of arguments can be a generator (specifically, an alternation expression). There are some situations in which this kind of formulation may provide a useful method of phrasing a computation. Consider, for example, a procedure parse() that parses a string. If the string cannot be parsed, the natural thing to do in Icon is for the procedure to fail. You might write something like this:

```
while s := read() do
  if not parse(s) then diagnose(s)
```

where diagnose() is a procedure that produces appropriate diagnostic information. You might decide it's more "Iconish" to rephrase this as:

```
while s := read() do
  parse(s) | diagnose(s)
```

(You might try using result sequences to see if these two formulations really are equivalent.) Better yet, phrase this computation as:

```
while s := read() do
  (parse | diagnose)(s)
```

If you think this kind of formulation is a bit odd, consider Alan Perlis' comment: "An idiom is a trick you use twice". Also, look at rsg.icn in the Icon program library. The syntax of the input language to this random sentence generator is designed around (and suggested by) applying a sequence of procedures to an argument until one succeeds.

Further Thoughts

We've only scratched the surface of what can be done with result sequences. We didn't even include limitation and bounded expressions. You might try adding these to the framework we've provided.

If result sequences appeal to you, think about how many phenomena in the world can be cast naturally in terms of sequences. You may be able to program in terms of sequences.

There's even an experimental programming language, called Seque, that is based on these ideas [3]. In Seque, result sequences are data objects that can be constructed and manipulated as first-class values.

Seque never made it beyond design and a prototype implementation (written in Icon). Maybe someday … .

References

1. *Control Mechanisms for Generators in Icon*, Stephen B. Wampler, Doctoral Dissertation, Department of Computer Science, The University of Arizona, 1981.

2. "Result Sequences", Stephen B. Wampler and Ralph E. Griswold, *Computer Languages*, Vol. 8, No. 1 (1983), pp. 1-13.

3. "Seque: A Programming Language for Manipulating Sequences", Ralph E. Griswold and Janalee O'Bagy, *Computer Languages*, Vol. 13, No. 1 (1988), pp. 13-22.

Procedure Libraries

Procedures provide one of the most powerful and convenient ways of adding functionality to Icon programs. Over the years, a large number of Icon procedures have been written for performing all kinds of tasks. Some of the most useful of these procedures are included as part of the Icon program library. (The remainder of the library consists of complete programs and data; we're only interested in the procedures here.)

The Icon program library is readily available at a nominal cost. Yet it seems that many Icon programmers don't have it or don't use it. We're reminded of this when we get a request to add a new function to Icon and find there's already a procedure in the Icon program library that does what's needed.

You might argue that a function, being built into Icon, is sure to be faster than a procedure that is written in Icon. That's true, but in most cases we've seen, the actual difference in efficiency is minor — and it's unquestionably *much* easier to use an existing procedure than it is to modify Icon itself.

Persons who use the Icon program library tell us how much easier it has made their Icon programming — and they often make their own contributions to the library. Why, then don't more Icon programmers use the library?

We think part of the reason is that it's not so obvious at first glance how to use the library. It takes a little effort to set things up so that the library is easy to use. And, perhaps more important, it takes a little digging to find out what procedures are in the library and how to use them.

In this article we'll try to "demystify" the Icon program library and show what's actually going on when you use it. Of course, you don't have to have the Icon program library to use the information that follows; you can write your own procedures to fit your own needs.

Incidentally, the Icon program library can be used with either the Icon interpreter or the Icon compiler. What actually happens in the two cases is somewhat different. Most of what follows applies to the interpreter, which most Icon programmers use. Later on in this article, we'll have something to say about how the library applies to the compiler.

Before going on, we need to explain a matter of terminology. We'll refer to a collection of procedures in a single source-code file as a *library*. To avoid confusion with the Icon program library, we'll abbreviate it to Ipl. (For you old timers, Ipl does not mean "initial program load"!)

Ucode and Icode

As described in the article on variant translators, *ucode* is created by the Icon interpreter as an intermediate step between a source-code and the *icode* that the interpreter actually executes. The origin of the words ucode and icode is obscure and not important. It's enough to think of ucode as the result of translating an Icon source file into virtual machine language for an imaginary Icon computer — much like assembly language for a real computer.

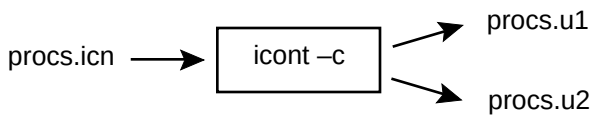
The reason for producing virtual machine code rather than real machine code is described in the reference at the end of this article and will be explained in more detail in an article in the next issue of the Analyst.

An icode file is essentially a compact "binary" form of virtual machine code that is suitable for execution by the imaginary Icon computer. The Icon interpreter emulates this imaginary computer.

Ucode files are text files; you can read or print one if you wish. Icode files, on the other hand, are not easily readable.

Translating an Icon source file (with suffix .icn) creates a pair of ucode files with suffixes .u1 and .u2. The .u1 file contains virtual-machine code for the procedures in the source file, while the .u2 file contains global information about the program. There is a pair of files, rather than one, since the translator does not produce information in the order needed to compose a single file and it would be complicated to combine the files later.

When an Icon program is translated, the ucode files normally are deleted and only the icode file is preserved. However, if you use the –c option to icont, the ucode files are preserved (and no icode file is created). The process of producing a pair of ucode files can be visualized as follows:



Although an icode file must contain a complete program, ucode files need not. This is the way the Icon interpreter supports separate compilation of program components. More precisely, sets of procedures can be organized into source-code files ("libraries" as defined above) and the corresponding ucode files can be combined with other ucode files to form a complete program in an icode file. In this way, procedures in a library can be incorporated into any Icon program that needs them.

Linking Ucode Files

Ucode files are *linked* to form an icode file. Linking can be done in two ways — by including the name of a ucode file pair on the command line when an Icon source file is translated or by means of a link declaration in the source program itself.

The command-line method has the form

```
icont prog.icn procs.u
```

where prog.icn is a source-language file that is linked with the ucode file pair for procs. The .u suffix distinguishes a ucode file pair (the corresponding .u1 and .u2 files) from a source-language file.

The link declaration, which is included in the source program that needs a ucode file, has the form

```
link name
```

where *name* is the name of a ucode file pair (without the .u). For example, prog.icn might start as

```
link procs

procedure main()
   .
   .
```

When this file is translated by

```
icont prog
```

the ucode file pair for procs is included automatically.

Several libraries can be included in a program by using several link declarations or by listing the ucode names separated by commas in a single link declaration, as in

```
link procs1
link procs2
link procs3
```

or

```
link procs1, procs2, procs3
```

While it may be handy during program development to specify ucode files on the command line, it's generally better style to use link declarations in the program itself. That way, the information about which libraries a program needs is contained in the program itself.

Locating Ucode Files

The Icon interpreter needs to be able to find the ucode files given in link declarations. It always looks first in the current directory, so you don't need to do anything special if you put your ucode files in the same directory as the programs that use them. However, in order to organize ucode files and to be able to access them wherever your programs are, you probably will want to place them in a special place.

The link declaration allows path specifications in place of simple file names. Such path specifications, which must conform to the path syntax of the computer system you're using, need to be enclosed in double quotes (as if they were string literals). An example for a UNIX system is

```
link "/usr/icon/ilib/procs"
```

Either complete path specifications, as in this example, or partial path specifications can be used.

Path specifications are tedious to write, and if you move your ucode files, all the source files that refer to them may need to be changed. To avoid such problems, the Icon interpreter uses the environment variable IPATH when looking for ucode files given in link declarations.

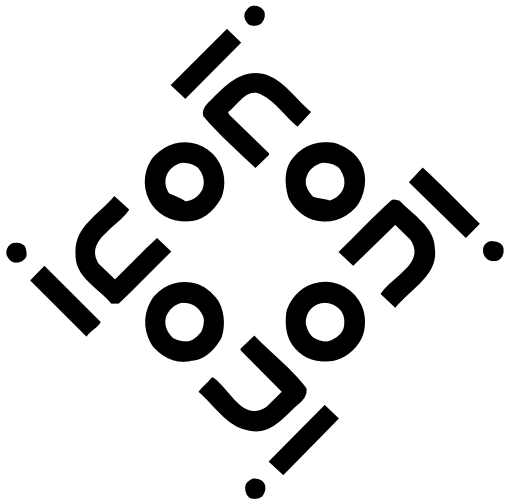
The value of IPATH is a blank-separated list of paths (partial or complete) of directories containing ucode files. For example, in BSD UNIX

```
setenv IPATH "/usr/icon/ilib /usr/icon/explib"
```

sets IPATH so that /usr/icon/ilib and /usr/icon/explib are searched (in the order given) for ucode files given in link declarations. With IPATH set, all that's needed in a link declaration is

```
link procs
```

where the ucode file pair for procs is in one of the directories



given by IPATH. It's worth knowing that the current directory is searched first, regardless of IPATH.

Name Collisions

If you're writing your own procedure libraries, there are some things you should consider. One is the possibility of name collisions.

When you write a self-contained Icon program, any conflicting uses of the same name by different procedures are resolved in the debugging process. When you link library procedures, however, you don't necessarily know the names of all the names used in the library procedures; in fact, if you only have ucode files but not the corresponding source files, you can't easily find the names used.

If the same name is used for a global identifier (such as a procedure name) in both your program and a linked library, the Icon linker notes the conflict and terminates with a fatal error, just as if you'd declared the same global identifier more than once in your program. It's undeclared local identifiers that cause the real problem.

An undeclared identifier is taken to be local, provided there is no global declaration for it. If there's a global declaration, however, the undeclared identifier is global. An undeclared identifier in a program that otherwise would be local becomes global if a linked library has a global declaration for it. This unexpected change in scope can be disastrous and mysterious. Consider, for example, a program that has code such as this:

```
procedure strfrm(i1,i2)
   complex := i1 || "+" || i2 || "i"
      .
      .
      .
```

Now suppose this program links a library that contains the record declaration

```
record complex(r,i)
```

The identifier complex is global by virtue of the record declaration. The use of complex in strfrm(), which presumably was intended to be local, is now global. Assignment to it in strfrm() wipes out the record constructor for complex, which presumably is needed in the linked library. Imagine the consequences.

The moral is clear — always declare all local identifiers.

For similar reasons, if you are writing procedure libraries, and need global identifiers that are not used outside of the library, it's a good idea to give them names that are not likely to conflict with names in programs that include the libraries. Mixed upper- and lowercase letters with interspersed underscores, as in Init_Table, is one convention used to minimize the likelihood of name collisions. Note, however, there is no way to guarantee there won't be a collision. At least with global identifiers, the linker detects the problem.

Incidentally, the procedures in the Icon program library do not follow any consistent rules to avoid name collisions, but all local identifiers in them are declared as such.

Organizing Libraries

Building a useful collection of ucode files is somewhat of an art. It requires experience and good taste.

The contending factors are functionality and the number of procedures per file (and hence ucode file pairs). When a ucode file pair is linked into a program, all the procedures in the ucode file pair are incorporated into the program. If there are 100 procedures and only one is needed, the other 99 are extra baggage, increasing linking time and the size of the

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

resultant icode file. On the other hand, if a program needs 100 procedures and each is in a separate ucode file pair, all 100 ucode file names must be specified in link declarations.

The usual compromise is to group procedures by functionality, tolerating the linking of a few unneeded procedures in some cases.

It's worth noting that libraries are not limited to procedures. They can include record and global declarations as suggested above, and they also can contain link declarations. Link declarations in libraries can be used to combine several other libraries "transparently". For example, a library may consist only of link declarations for other libraries. Suppose math.icn contains

```
link integer, real, rational, complex
```

then

```
link math
```

could be used to link the four other libraries.

There's no end to what you can do with this technique if you have the energy and discipline to keep track of everything.

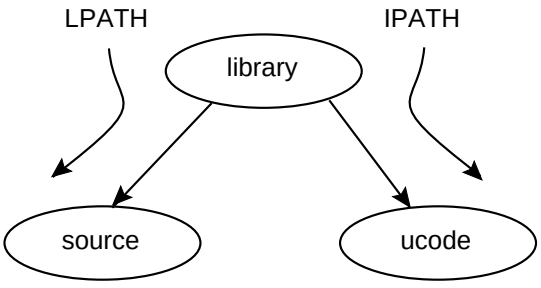
Using Libraries with the Icon Compiler

Libraries are used somewhat differently in the Icon compiler, although it's transparent in source programs that use libraries.

Ucode files don't mean anything to the compiler, which produces code for a real computer (via C) instead of code for an imaginary computer. Instead, the compiler supports link declarations by including source code instead of ucode. The link declarations are the same; the compiler just looks for source code.

To allow the interpreter and the compiler to be used for the same source program, the compiler uses a different environment variable, LPATH, to find the source code for library programs given in link declarations. Other than looking for source code rather than ucode, LPATH works the same way in the compiler as IPATH does in the interpreter.

If you're using both the interpreter and compiler for the same program, (the interpreter for program development and the compiler for the final executable program, for example), it may help to keep things straight if ucode and source code files are kept in parallel directories, as in



What's in the Icon Program Library

As mentioned earlier, if you're using procedure libraries with your Icon program, the Ipl is a good place to start; a lot of effort has been put into the procedures there and you might as well take advantage of them.

The Ipl consists of a basic part and updates. Updates, which contain corrections, improvements, and new material, are issued about three times a year. You can get the basic library in a variety of ways. The updates are provided on a subscription basis. See any recent *Icon Newsletter* for details.

To date, there are 70 procedure libraries in the Ipl, comprising a total of 300 procedures. See the box at the bottom of this column for a list of what some of these libraries do.

Reference

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

Typical Procedures in the Ipl

Arrange data in columns
Collate and decollate strings
Control ANSI terminal
Convert between ASCII and EBCDIC
Convert hexadecimal numbers
Convert number formats
Copy files
Encode/Decode structures as strings
Expand wild-card characters in file names
Find regular expressions
Generate n-grams
Match patterns
Perform complex arithmetic
Perform radix conversion
Perform rational arithmetic
Permute characters
Process command-line options
Produce string images of structures
Quote strings for shell commands
Scan lists
Segment strings
Shuffle elements
Simulate n-tuples
Snapshot string scanning
Wrap text lines
Write in C-style printf form

Corrections

Here are a couple of corrections to Issue 4 of the Analyst, as provided by readers.

At the bottom of page 3 and the top of page 4, Paul Abrahams notes that the first instances of tab(many(&letters)) should be tab(upto(&letters)) as in the preceding examples.

Mark Emmer notes that the amount of space needed for large integers as given on page 6 is misleading. He provides this correction:

Large integers have 18 bytes of overhead on a 16-bit integer system and 20 bytes of overhead on a 32-bit system. The "digits" for large integers are 2 bytes each. Here's the amount of spaced used on 16-bit systems for some large integers of the form 10^n:

n	digits	bytes	ratio
10	3	24	0.42
20	5	28	0.71
50	11	40	1.25
100	21	60	1.67
500	104	226	2.21
1000	208	434	2.30
10000	2077	4172	2.40

This is as expected, tending toward 2.40824 decimal digits per byte ($2.40824 = 8 \times \log_{10}2$).

The Analyst had it backwards, saying "you can count on less than two bytes per decimal digit for really large integers." It should be "less than 1/2 byte per decimal digit".

If you have ProIcon 2.0 for the Macintosh, there are some nifty things you can do with resources. Mark Emmer provides these tips:

Dialog boxes used with the message() and gettext() functions can be customized to include additional static text, screen icons, or pictures.

You can do so by copying these dialog boxes from the ProIcon application to the resource fork of an Icon program file with Apple's ResEdit program. The necessary resources to copy are:

- For a modal message() (with OK button): DLOG 415, DITL 415
- For a non-modal message() (that is, a non-modal display of text without an OK button): DLOG 416, DITL 416
- For gettext(): DLOG 418, DITL 418

Once these resources are copied to the Icon program file, they can be modified by ResEdit. Do not delete or renumber any of the standard elements present in the dialog (text boxes and buttons). It is permissible to reshape them and move them around. You can also add additional elements such as screen icons, additional static text, and PICTs.

During execution, dialog resources in the Icon program supersede those in the ProIcon application. Resources may be provided in linked files as well as in the Icon main program. Note that linked files may contain resources and no program text at all!

If you want to use several different versions of each dialog in a program, it is necessary to dynamically switch between the resource forks of the files containing the different versions. Use

```
id := callout("CODE", "CurResFile")
```

to obtain and remember the current resource file ID.

```
resFile := callout("CODE", "OpenResFile", filename)
```

can be used to open each of the files containing the desired dialogs.

```
callout("CODE", "UseResFile", resFile)
```

can then be used to switch between files prior to calling message() or gettext(). CurResFile and UseResFile are described the README file in the External Functions folder in the ProIcon 2.0 distribution.

What's Coming Up

In the next issue of the Analyst we'll have procedures for the string synthesis facility described in this issue. You have two months to write your own before seeing ours.

We'll also start exploring the imaginary computer that provides the conceptual framework for the implementation of the Icon interpreter and have the first of a series of articles on the optimizing compiler for Icon.