The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1992
Number 12

In this issue …

Exercises … 1
Anatomy of a Program … 2
Inside the Icon Compiler … 4
Programming Tips … 9
Looking Ahead … 11
Reader Feedback … 12

Exercises

We're starting this occasional feature of the Analyst to pose some programming problems on which you can try your Icon programming skills. We'll provide our solutions in the next issue of the Analyst.

In an earlier article on result sequences [1], we suggested using sequences of values as a conceptual tool and programming technique. If you're going to use sequences, you'll need skill in writing them. Otherwise you'll be distracted from the *concept* of sequences by having to figure out how to produce them.

Here are some sequences for which you should write generating expressions. To make these a bit more interesting, you're not allowed to use procedures or co-expressions in your formulations. You may use variables; in fact, you'll have to use variables in some cases.

1. A sequence consisting of the names of the months of the year: "January", "February", … "December".

2. A sequence consisting of the lowercase letters in increasing alphabetical order.

3. A sequence consisting of the lowercase letters in decreasing alphabetical order.

4. An infinite sequence consisting of the lowercase letters in increasing alphabetical order, repeatedly.

5. An infinite sequence consisting of the lowercase letters in decreasing alphabetical order, repeatedly.

6. A sequence consisting of strings representing the times in minutes in the 24-hour day, starting midnight and ending at the minute before midnight: "00:00", "00:01", … "00:59", "01:00", … "23.59".

7. An infinite sequence consisting of the digit 1: 1, 1, 1, 1, … .

8. An infinite sequence of randomly distributed strings "H" and "T".

9. An infinite sequence consisting of randomly selected digits.

10. An infinite sequence consisting of randomly selected characters.

11. An infinite sequence consisting of the squares of the positive integers: 1, 4, 9, 16, … .

12. An infinite sequence consisting of the Fibonacci numbers: 1, 2, 3, 5, 8, 13, 21, 34, … .

13. An infinite sequence consisting of the factorials of the positive integers: 1, 2, 6, 24, 120, 720, … .

14. An infinite sequence consisting of the "triangular numbers": 1, 3, 6, 10, 15, 21, … .

15. An infinite sequence consisting of the prime numbers: 2, 3, 5, 7, 11, 13 … .

16. An infinite sequence consisting of *n* copies of each positive integer *n*: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, … .

To make these exercises a little more fun, try to write the most concise solutions you can. Having done so, you might think about whether the most concise solutions really are the best. Are they faster than longer solutions? Are they easier to understand? Are they aesthetically more pleasing?

Incidentally, there's a fascinating book on integer sequences [2]. If you are interested in this kind of thing, check your local library. The book may even still be in print.

References

1. "Result Sequences", The Icon Analyst 7, pp. 5-8.

2. *A Handbook of Integer Sequences*, N. J. A. Sloane, Academic Press, 1973.

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

Anatomy of a Program — A Suffix Calculator

This is our second adventure in describing in some detail the workings of a complete Icon program. As in our first such article [1], we started with a program that we thought was in good shape. And wound up revising it several times before this article was finished.

The program here, a simple suffix calculator for Icon expressions, illustrates the use of stacks, error conversion, and some fine points of string invocation.

The Specification

The suffix calculator is designed to accept lines of input as values: integers, real numbers, csets and strings, all in the style of Icon literals, with an empty line standing for the null value. Lines of input that do not satisfy this syntax are either Icon operators, Icon functions, or calculator commands. For example, the lines of input

```
"abc"
3
repl
write
```

cause abcabcabc to be written and that value left on the top of the stack.

Icon operator symbols can be given also, so that

```
4
2
+
```

leaves 6 on the top of the stack.

Icon has unary (prefix) and binary (infix) operations that use the same symbol. The calculator takes such a symbol as representing the binary operator and the unary operator is not available. For functions like write() that can take an arbitrary number of arguments, the calculator assumes they take one argument. See the exercises at the end of this article for extensions to generalize the handling of these kinds of situations.

There are three calculator commands:

- dump — print all the values on the stack
- clear — remove all values from the stack
- quit — exit from the calculator program

In order to make the calculator robust, errors are converted to failure. Both ordinary expression failure and failure as a result of errors produce diagnostic messages and leave the stack unchanged.

The Program

The program for this suffix calculator is quite short and is shown in on the next page.

❶ The main procedure starts by creating a stack in the form of an empty list. The rest is just a read-and-process loop.

❷ As noted above, there are three possibilities, depending on what a line of input is: values, operations (including functions), and commands. Anything else is an error.

In many programming languages, the processing would be cast in an if … else if… else if … else construction. While that can be done in Icon, alternation is more natural, with the evaluation of procedures proceeding until one succeeds:

```
value(line) | operation(line) | command(line) |
  Error("erroneous input", image(line))
```

Recalling that procedures applied to a common argument can be factored out [2], this can be cast more concisely as

```
(value | operation | command)(line) |
  Error("erroneous input", image(line))
```

The first three procedures could be given in any order, since the syntax they require is mutually exclusive. It's most likely that values and operations will be used more frequently than commands, so command() is placed last for reasons of efficiency. It's hard to chose between values and operations; either could come first without much likely difference in efficiency.

❸ Continuing alphabetically with the procedures, the implementation of command() is straightforward.

The command clear just creates a new, empty stack — one of the advantages of having run-time creation of structures in Icon. Note that stack is a global variable so that all the procedures can access it.

The command dump also is simple. Note the use of image() to allow, for example, the integer 4 to be distinguished from the string 4.

The command quit causes program execution to terminate with a normal completion code. Note that as discussed in Reference 3, the use of stop() would be inappropriate here. Note also that while the user could enter exit as a function, exit() expects an integer argument that the user would have to enter first.

❹ Note that command(line) must fail if line is not a command — this drives the alternation in the main processing loop.

❺ Operations are a bit more challenging. String invocation can be used to invoke a function or operator by its string name as it is input to the program. proc() converts a string to the corresponding function or operator, if the string represents the name of one, but fails otherwise. However, in the case of the string name of an operator, proc() requires a second argument that specifies whether the operator is unary, binary, or ternary (to-by, whose name for the purposes of string invocation is "..." is one of the two ternary operators — we'll let you identify the other one).

```
link escape, usage

global stack

procedure main()
  local line

  stack := [ ]                                                       ❶

  while line := read() do
    (value | operation | command)(line) |                            ❷
      Error("erroneous input ",  image(line))

end

procedure command(line)

  case line of {                                                     ❸
    "clear":  stack := [ ]
    "dump": every write(image(!stack))
    "quit":    exit()
    default:  fail                                                   ❹
    }

  return

end

procedure operation(line)
  local p, n, arglist

  if p := proc(line, 2 | 1 | 3) then {      # function or operation?  ❺
    n := abs(args(p))
    arglist := stack[–n : *stack + 1] | {                            ❻
      Error("too few arguments")
      fail
      }
    stack := stack[1 : –n]
    &error := 1                             # anticipate possible error ❼
    put(stack, p ! arglist) | {             # invoke                  ❽
      if &error = 0 then                                             ❾
        Error("error ", &errornumber, " evaluating ", image(line))
      else
        Error("failure evaluating ", image(line))
      stack |||:= arglist                   # restore unused arguments ❿
      }
    &error := 0                                                      ⓫
    return
    }
  else fail                                                          ⓬

end

procedure value(line)                                                ⓭

  put(stack,
    2(line == "", &null) |                                           ⓮
     numeric(line) | {                                               ⓯
       line ? {
         2(="\"", escape( tab(–1)), "\"" ) |                         ⓰
         2(="'", cset(escape(tab(–1))), ="'" )
         }
      }
    ) | fail

  return

end
```

The expression

proc(line, 2 | 1 | 3)

succeeds for the first value (2, 1, or 3) for which line is the name of an operation. 2 is tried before 1 to select the binary operator when there also is a unary one with the same name. 3 is last simply because it is least likely. You may be wondering what happens if line is the name of a function. If its first argument is the name of a function, proc() ignores its second argument; handy here.

If proc() succeeds, it assigns the corresponding function or operator (a value of type procedure, not string) to p, from which the number of arguments can be obtained using args(). There's still a wrinkle — args() produces –1 for functions like write() that can take an arbitrary number of arguments. Hence the use of abs().

❻ Now we're ready to apply the operation to its arguments. The stack is split into two parts for this. Although we've not gotten that far, note that for this to be done conveniently, the top of the stack needs to be at the right end of the list.

❼ Before invoking the operation, &error is set to 1 so that an error will not cause the calculator to terminate.

❽ Now everything is ready to do the actual invocation. This is just what list invocation is designed to do easily [4]. If the invocation succeeds, the resulting value is put on the stack. Normally, we'd use push(), not put() for stack manipulation. But as noted above, it's convenient for the arguments to be in left-to-right order on the stack and hence to have the top of the stack at the right end of the list. Although we don't need it here, pull() in combination with put() provides the same stack functionality as pop() in combination with push().

❾ If the invocation fails, it could be because of failure of the evaluation of the expression, or it could be because an error occurred. The two cases can be distinguished by checking the value of &error, since

it is automatically decremented in case an error is converted to failure.

⑩ If the invocation failed (for whichever reason), the arguments are restored to the stack by concatenation.

⑪ Finally, &error is reset to 0 to avoid its masking some other error in the program.

⑫ If proc() fails at the beginning of the procedure, then operation() also fails.

⑬ The final procedure, value(), has three basic cases to consider: an empty line, indicating the null value, a number, or a quoted literal.

⑭ The second argument to put() is cast as an alternation for these three cases. In the case where an expression can fail, mutual evaluation is useful, since if any expression in it (such as line == "") fails, the entire mutual evaluation fails and the next alternative is tried.

⑮ The function numeric() handles both integers and real numbers, returning the correct type.

⑯ Quoted literals are analyzed using string scanning as you'd expect. Basically, it's a matter of checking the first and last characters to see if they are quotes of the same kind. If so, the characters between them give the desired value. The procedure escape(), linked from escape.icn in the Icon program library, takes care of escape sequences. This allows the user of the calculator to enter characters in string and cset literals that can't be keyboarded directly, such as "\n".

The procedure escape() is rather involved because of the several kinds of escape conventions that Icon supports in quoted literals. It also has to take into account differences between ASCII and EBCDIC in some escape conventions. If you're interested in the gory details, see the Icon program library.

This program uses another procedure from the Icon program library, Error(). It shows an interesting wrinkle in the use of list invocation. Error() is designed to accept an arbitrary number of arguments, which it writes in order to standard error output. It's declared with a single list argument:

```
procedure Error(L[ ])
  push(L, "*** ")
  push(L, &errout)
  write ! L
end
```

The interesting point is the way that a string of asterisks and &errout are pushed to add two arguments to the list passed in the call of Error() prior to the invocation of write().

Retrospective

As we said earlier, writing this article led to several improvements to the program. If you have an old version of the Icon program library, compare calc.icn to the version given here to see how much we changed. In fact, writing a detailed description like this seems to be a guaranteed way to improve a program.

There are some enhancements we didn't make that you might want to try as exercises:

• Provide a way of supporting both the unary and binary operations that have the same operator symbol.

• Provide a way of specifying how many arguments a function like write() should take.

• The calculator, as it stands, ignores generators, using only the first value produced even if the operation could produce many. Modify the program so that all the values produced by a generator are put onto the stack. Rethink the handling of expression failure in this context.

• If all the values a generator can produce are put on the stack, there's the possibility of an infinite number, or even an inconveniently large finite number. Provide a way for the user to limit the number of results that are put on the stack as the result of generation.

• Provide a way to prevent a user from specifying a procedure (as opposed to a function). While you may imagine uses for procedures, they inevitably are part of the calculator itself, and they probably should not be accessible to a user; or at least to a casual user.

Reference

1. "Anatomy of a Program — A Recognizer Generator", The Icon Analyst 10, pp. 4-9.

2. "Result Sequences", The Icon Analyst 7, p. 7.

3. "Program Termination", The Icon Analyst 6, p. 11.

4. "A String Evaluator", The Icon Analyst 9, p. 3.

Inside the Icon Compiler

Now that there's an optimizing compiler for Icon, we have lots of things to write about: how it works, what it does, how to use it, and so forth.

We won't attempt to get deeply into the compiler in these articles in the Analyst — the compiler is a sophisticated and complex beast and it takes quite a bit of background and study to even become generally familiar with how it works. Hitting the high spots here and there may, however, prove both interesting and useful.

Most of the material here is adapted from Ken Walker's doctoral dissertation [1].

Components of the Compiler

The diagram at the bottom of the opposite page shows what goes on when the Icon compiler is used. The four components that constitute the Icon compiler system are shaded:

• The compiler itself.

• A data base that contains information about Icon that the compiler needs in order to perform optimizations and generate code.

• C header files that are needed by the C code that the Icon compiler produces.

• A library of object code that is linked with the object code produced for the program being compiled.

The Icon compiler is simply (!) a program written in C. More on it in a moment. But what about the data base and object library? They are produced when the Icon compiler system is built, as shown in the diagram on the next page.

The key to this part of the compiler system is the run-time source code. This code consists of the routines and data that make up Icon's run-time system: code for its functions, operations, keywords, and things like storage management. Most of the semantics of Icon reside in this run-time source code. (The major exceptions are the semantics of control structures, which are embodied in the compiler itself.) The compiler relies on information in a data base derived from the run-time source code for the properties of functions, operations, and keywords.

Prior to the development of the Icon compiler, the run-time system for the Icon interpreter was written in C. The compiler, however, needs information about functions, operations, and keywords that is not easily extracted from C code. To accommodate this need, a special run-time language, RTL, was designed for the compiler's run-time system.

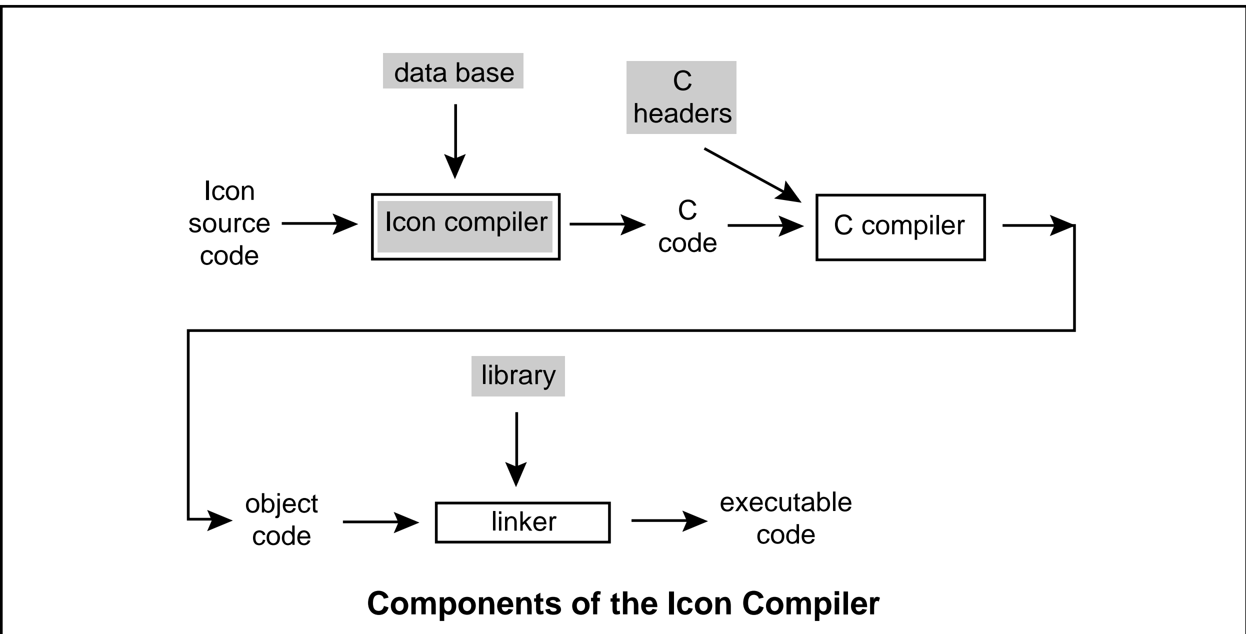
RTL is a superset of C and provides constructs for describing the semantics of Icon as well as ways for expressing operations like type checking and conversion in a more convenient way than is possible in C alone. We'll have more to say about RTL in a subsequent article, but you can get a hint of what it's like by looking at the box at the right.

```
"numeric(x) – produces an integer or real "
"number resulting from the type conversion of x, "
"but fails if the conversion is not possible."

function{0,1} numeric(n)

   if cnv:(exact)integer(n) then {
      abstract {
         return integer
         }
      inline {
         return n;
         }
      }
   else if cnv:real(n) then {
      abstract {
         return real
         }
      inline {
         return n;
         }
      }
   else {
      abstract {
         return empty_type
         }
      inline {
         fail;
         }
      }
end
```

An Example of RTL Code



Components of the Icon Compiler

The run-time translator processes RTL code and produces two things: corresponding C code and the data base mentioned above.

The C code produced by the run-time translator is then compiled along with support routines written in C to produce object code, which in turn is processed by a librarian to produce the object-code library. This object code library is linked with the object code that results from compiling an Icon program.

All this sounds simple, and it is, at least conceptually. In practice, building the data base and the library is a complicated and time-consuming process. Once they are built, however, they become part of the Icon compiler system and there's no need to rebuild them unless Icon's run-time system is changed (as, for example, the result of adding a new function or keyword to Icon).

The Compiler Itself

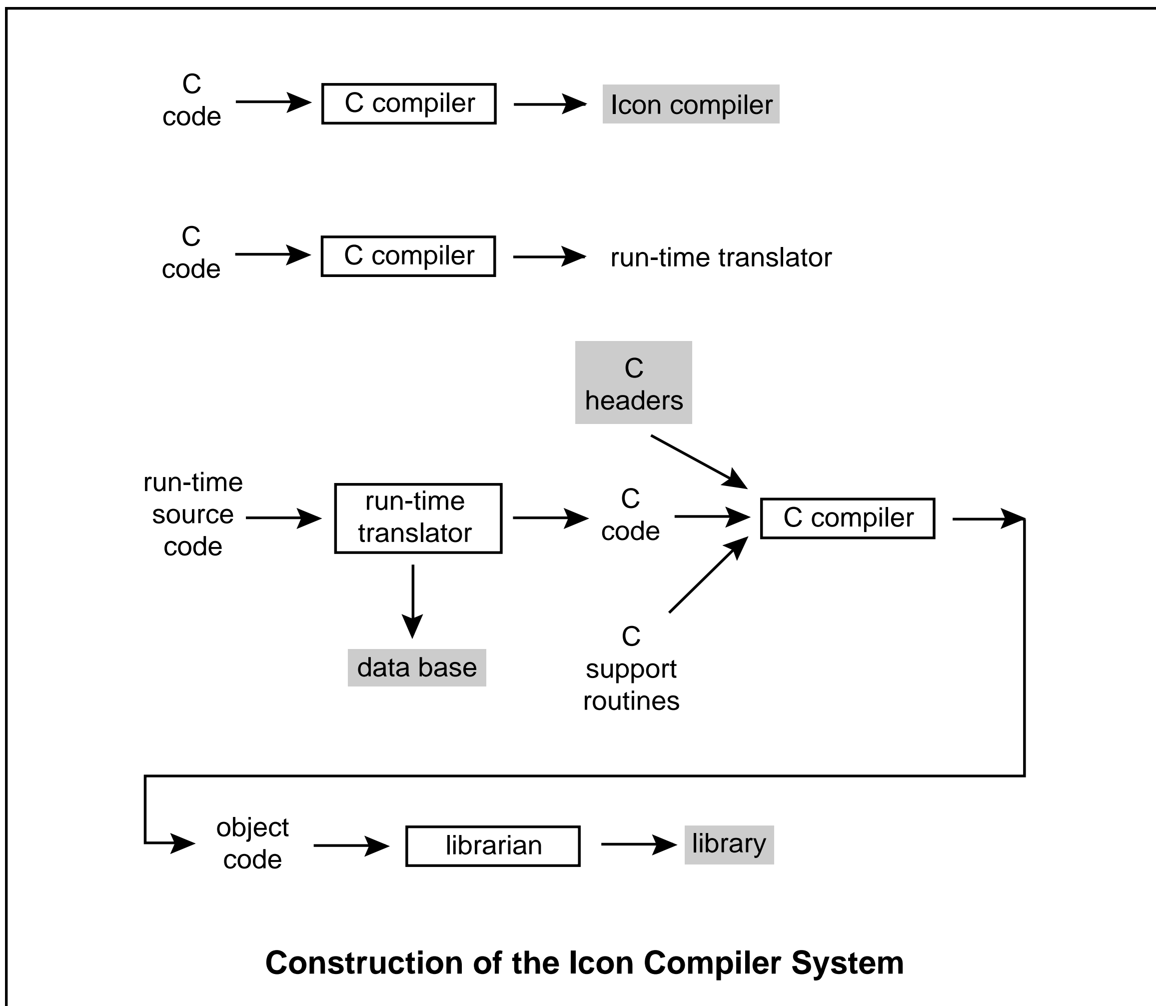
As shown in the diagram on page 5, there are two sources of input to the Icon compiler: the Icon source program it compiles and the data base mentioned above. There are several phases in the compilation process:

- initialization
- source-code analysis
- symbol resolution
- naive optimization
- type inference
- code generation
- invocation of the C compiler

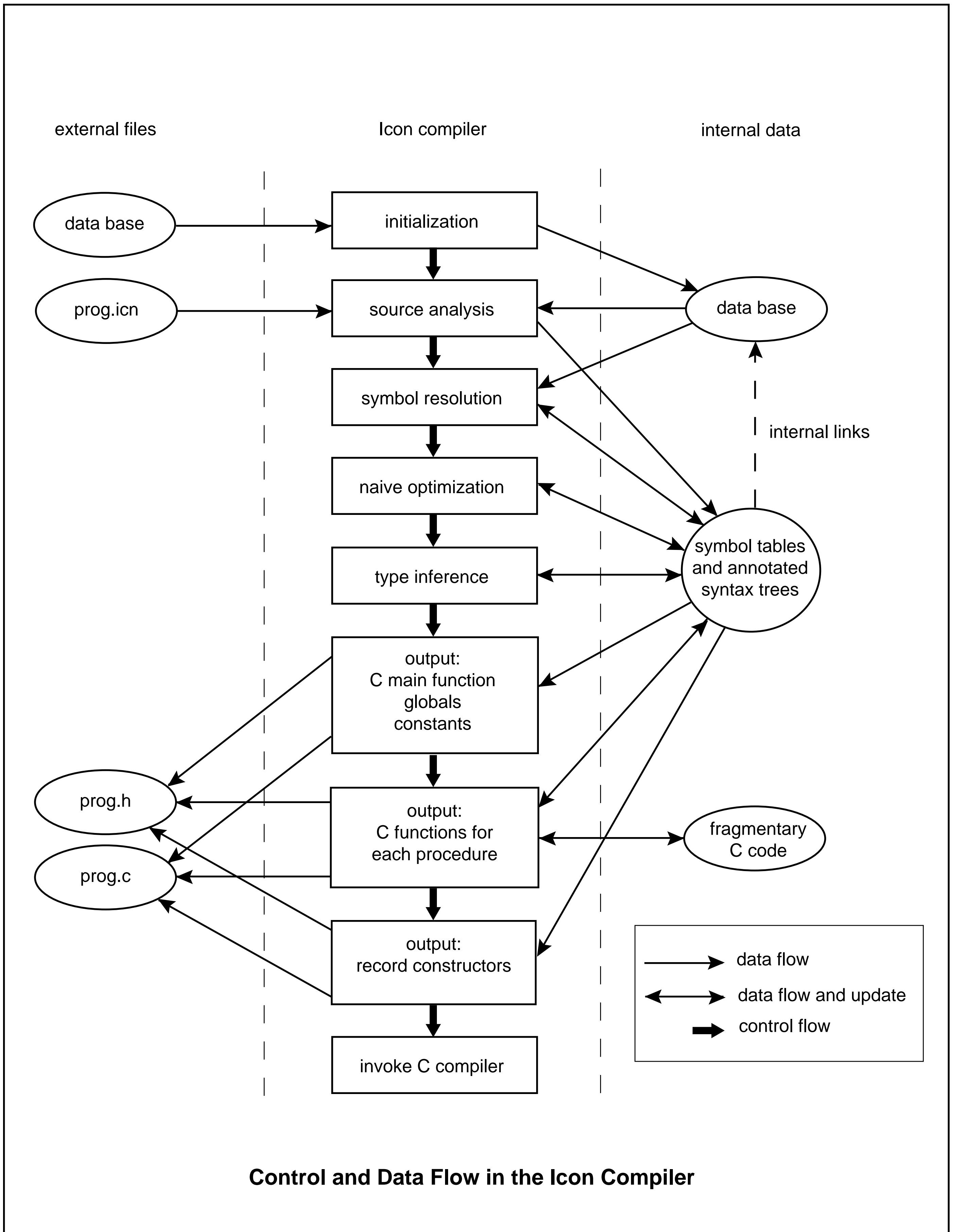
The control and data flow among these phases are shown on the next page. External files are shown at the left: The data base and the Icon program being compiled are input; the C program and header files for it are output. Data stored in memory is shown at the right.

The initialization phase reads the data base into memory. The source-analysis phase consists of a lexical analyzer and parser similar to the ones used in the Icon interpreter. The parser generates abstract syntax trees and symbol tables for all the procedures in the source program. See the diagram on page 8.

The symbol resolution phase determines the scope of undeclared identifiers much in the manner of the linker for the Icon interpreter [2].



Construction of the Icon Compiler System



Control and Data Flow in the Icon Compiler

Naive optimizations related to invocation and assignment are performed next. As described in Reference 2, functions and procedures are the initial values of global variables. Unless assignments are made to these variables, function and procedure invocations "are what they seem to be". The Icon compiler checks for possible assignments to these variables. If there are none, it binds their invocations to the actual functions and procedures.

The naive optimization for assignment deals with the common case where an unconditional assignment is made to a named variable. By identifying such assignments, subsequent optimizations are easier to perform and are more effective.

Type inference is the "biggy". We hinted at the kinds of things it does and why they are important in Reference 3.

Actual code is produced on a per-procedure basis. It involves several sub-phases as shown in the diagram on the next page:

- liveness analysis
- code generation
- fix-up and peephole optimization
- output

Liveness analysis involves determination of the life times of intermediate computations so that temporary memory locations can be assigned to them. This process is more complex in Icon than in most other programming languages because goal-directed evaluation can cause backtracking and the re-use of previously computed results. See Reference 4 for a discussion of this problem.

Before any output is actually done, fragmentary C code is constructed in memory. Portions of the code are easier to complete after other parts are produced. If this code were written out as-is, it would contain many unnecessary instructions because code segments produced for separate portions of a program usually do not fit together well. These problems are handled by fix-up routines and a peephole optimizer.

Finally, the actual code is written. For convenience, there are two output files, one that includes the other as a header file.

Conclusion

We've glossed over many aspects of the compiler and deliberately over-simplified some issues to try to give a general feeling of what goes on inside the compiler.

If you're interested in knowing more, there are two sources of detailed information: Ken Walker's dissertation [1] and the source code itself, which, as always, is the "final authority".

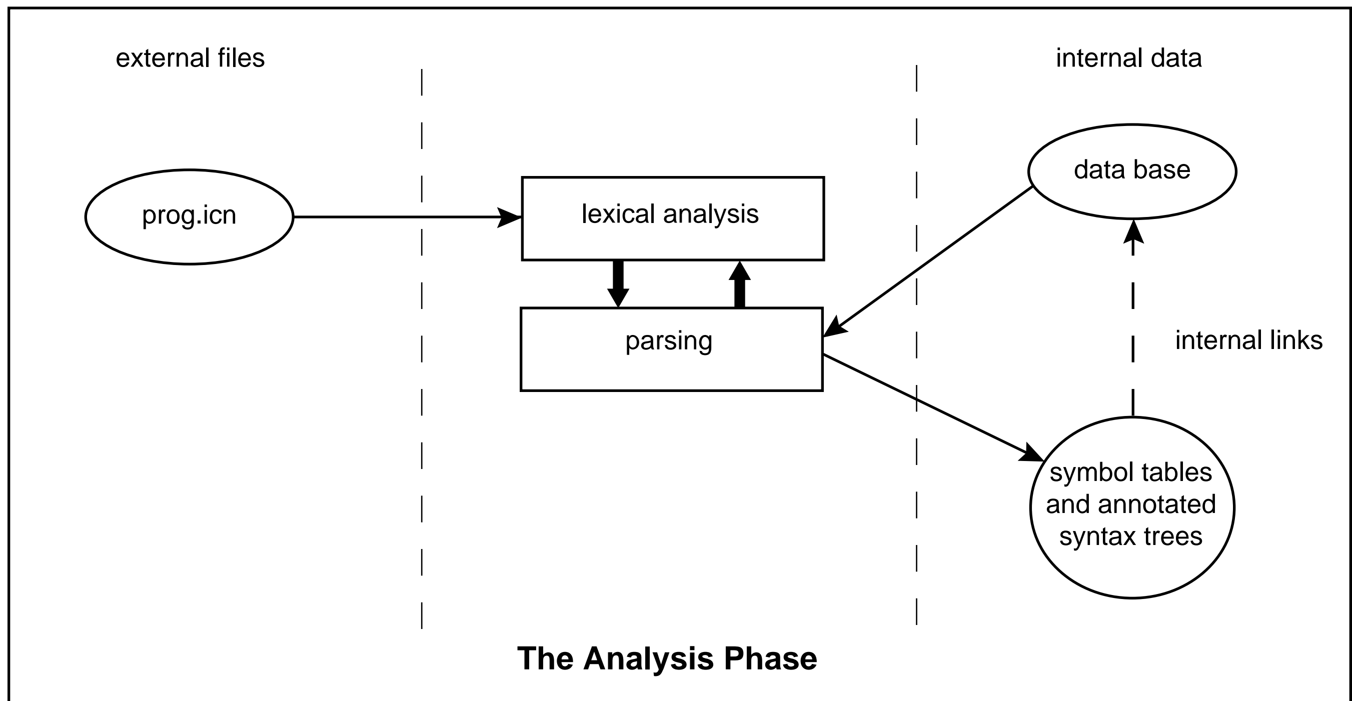
References

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

2. "An Imaginary Icon Computer", The Icon Analyst 8, p. 4.

3. "Type Inference in the Icon Compiler", The Icon Analyst 9, pp. 7-11.

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



The Analysis Phase

external files

internal data

liveness analysis

code generation for a procedure

fix-up and peephole optimization

output

symbol tables and annotated syntax trees

fragmentary C code

prog.h

prog.c

Code Generation

Buffering Input and Output

Reading and writing files are pretty much irreversible actions. Yes, you can close and reopen a file to read it again and use random-access input and output to get to a specific place in a file, but these are awkward and complicated operations compared to just reading and writing.

There are some situations, however, in which the natural thing to do is to write several lines, but not until some condition is satisfied. Similarly, sometimes when you read a line, you aren't ready to process it, although you may not know that until you've read it.

You can contrive ways of handling these situations, storing data in variables or lists until you are ready to look at it or write it out. A more general and flexible method is to provide input and output buffering and procedures for managing the buffered data. Since Icon lists can be accessed as stacks and queues and because they grow and shrink automatically, it's very easy to provide such buffering facilities.

Here are some simple procedures that illustrate useful techniques. We'll assume only standard input and standard output; you can generalize these procedures to handle other streams.

We'll start with a procedure Read() that acts much like read(), expect that it keeps lines in a buffer. It starts out by creating a list with one line of input. Subsequently it reads another line into the buffer before returning the previous line:

```
global buffer_in, Eof

procedure Read()

  initial {
    /buffer_in := [ ]
    put(buffer_in, read()) | (Eof := 1)
    }

  put(buffer_in, read()) | (Eof := 1)

  return get(buffer_in)

end
```

A couple of things are worth noting here. When the end of the input stream is encountered, read() fails. That simply means no line gets added to the buffer. Eof is set so that an actual end-of-file condition can be checked. The next call empties the buffer, and when the buffer is empty, Read() fails. Note that it's safe in Icon to attempt to read from a stream after its end. If this were not true, this procedure would be considerably more complicated.

By having a buffer, it's possible to "look ahead" at the next line without removing it from the buffer:

```
procedure LookAhead()

  return buffer_in[1]

end
```

Note that this procedure fails if the buffer is empty.

It's also possible to "put back" lines to be "read" subsequently.

```
procedure PutBack(s)

  push(buffer_in, s)

  return

end
```

Of course, these lines need not be ones that have actually been read.

Note that PutBack() increases the number of lines in the buffer. You might want to modify Read() so that it only does actual input when the buffer is empty. Or you might not — see the example at the end of this article.

Output buffering is somewhat different, depending on the functionality you want. For example, if you don't want any lines to be actually output until you explicitly request it, the following procedure might be useful:

```
global buffer_out

procedure Write(s)

  initial buffer_out := [ ]

  push(buffer_out, s)

  return s

end
```

We'll leave it to you to extend Write() to handle an arbitrary number of arguments. You might also want to think about type checking.

To write out the lines in the buffer, all that's needed is:

```
procedure Flush()

  while write(pull(buffer_out))

  return

end
```

And you can get back lines that haven't actually been written:

```
procedure GetBack()

  return get(buffer_out)

end
```

Or you just might want to clear out what's in the buffer:

```
procedure ClearOut()

  buffer_out := [ ]

  return

end
```

You no doubt can think of other useful procedures. For example, there are situations in which you might want to have a minimum number of lines in the input buffer. This can be done with

```
procedure ReadAhead(i)

  initial /buffer_in := [ ]

  while *buffer_in < i do
    put(buffer_in, read()) | {
      Eof := 1
      fail
      }

  return

end
```

Here's a program that uses this feature. It writes only the last few lines of the input file:

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

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

```
procedure main(arg)
  ReadAhead(integer(arg[1] | 10))
  while /Eof do
    Read()
  while write(Read())
end
```

As we suggested earlier, a good buffering facility should be able to handle streams other than standard input and output. Here's a hint for how you might handle multiple streams:

```
record buffer(name, in, out)
```


The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

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

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

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...uunet!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

© 1992 by Madge T. Griswold and Ralph E. Griswold
All rights reserved.

Looking Ahead

With the next issue, the Analyst begins its third year of publication. This anniversary provides an occasion to take stock, to reflect, and to plan.

There's a questionnaire on the next page that we hope you will complete and send to us so that we'll have an idea of how we're doing and what changes we should make.

One thing we have in mind is to place more emphasis on what's new and what's in the works. Many of the articles in past issues of the Analyst describe the end results of research. Icon itself is not just the result of an effort to design a new programming language. Instead, Icon is a by-product of research, embodying the results of investigations into new ways to express nonnumerical computation.

Of course, by the time Icon grew out of these origins, was made into a complete language, implemented, and packaged for general consumption, the research behind it was largely obscured.

In upcoming issues of the Analyst, we're going to try to get a little closer to recent research and research in progress. This will mean material about some things that may not work out or that may work out differently from what we expect. Some of the things we plan to write about may not be available for public consumption for some time, if ever.

Despite the resulting lack of "present reality", we hope you'll find some of the things we're doing to be interesting. And maybe you'll have some ideas for us.

Reader Feedback

Some of you have commented from time to time on the content of the Analyst, but we'd like to hear from more of you and get your opinions on specific issues.

Please take some time to fill out the questionnaire that follows. Send it to the Icon Project at the address given in the publication box on page 11. You can fax it if you prefer — or even send electronic mail if that's easier for you. Although we'd like to get as many completed questionnaires as possible, any other form of feedback is welcome also.

We've tried to design the questionnaire so that you can respond in the manner you think best — we're interested in general reactions and nuances, not numerical scores. Please be as informative as you can, but don't get frustrated and give up because it takes too much effort to answer a question; we'd rather have a partially completed questionnaire than none.

We'll summarize the responses we receive in a future Analyst. Individual responses will be kept in confidence, of course. (If you want to be quoted, let us know — and be sure to add your name.)

Are you generally satisified with the content of the Analyst?

Do you find the Analyst useful?

If so, in what ways?

Is the general technical level of the material about right for you, or is it too high or low?

What *kinds* of articles do you like best?

What specific articles did you like most?

What *kinds* of articles do you like least?

Are the **Programming Tips** useful to you?

What is your profession?

For what do you use Icon?

On what platform(s) do you run Icon?

Is there any thing else you'd like to pass on? (Use additional sheets as needed.)