A Tutorial on Creating Run-time Operations for the Icon Compiler

Kenneth Walker

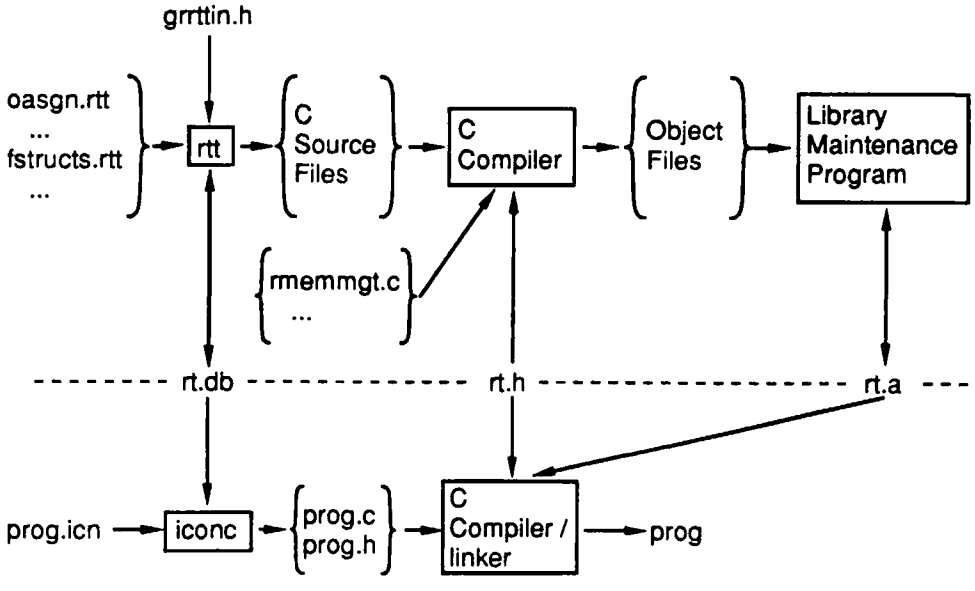
Department of Computer Science, The University of Arizona

1. Introduction

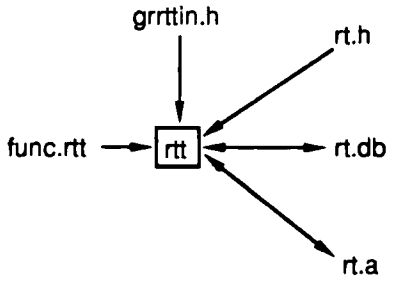
The Icon compiler [1,2] includes a run-time system that implements Icon's operators, built-in functions, and keywords and includes support routines for various features of the language. iconc, the compiler itself, contains no hard-coded information about built-in functions or keywords. Information about these kinds of operations is obtained at compile time from one or more data bases. New functions and keywords are easily added to the language without rebuilding iconc. This tutorial explains how to create these kinds of operations and make them available to the compiler.

In the compiler, Icon operations are written in a special implementation language. This language is built around C and has special features to support Icon. It is translated into pure C by the program rtt. A complete description of the language can be found in [3] with complete usage instructions for rtt in an appendix of that document.

The following diagram shows the overall structure of the Icon compiler. The portion of the diagram above the dotted line represents the run-time system and the portion below the dotted line represents the compiler itself. The arrows show data flow, with double headed arrows indicating that a program updates a file. The files ending with .rtt at the upper left of the diagram contain operation definitions. rt.db is the data base of operation information. rt.a is a link library of run-time operations and support routines. Files ending in .c and .h are ordinary C source files.



rtt automatically invokes the C compiler and the library maintenance program. The person adding run-time operations to the compiler has a simplified view of the run-time system:



This tutorial is an introduction to writing operations for the Icon compiler. It consists of four sections. The first section describes how to build and use data bases and libraries for the Icon compiler. The second section describes how to make existing C libraries available to Icon as built-in operations. The third section describes issues and techniques for writing new Icon operations from scratch. The final section presents suggested exercises to give the reader experience creating a new data base and library for the compiler.

Advanced information and techniques can be found in the references listed at the end of the tutorial and by studying the existing run-time system. The Icon implementation book [4], in spite of describing the interpreter rather than the compiler, is a particularly valuable reference.

2. Establishing Alternate Data Bases and Libraries

As shown in the first diagram in the introduction, the interface between iconc and the run-time system consists of three files: a data base, a C include file (that contains #include directives for other files not shown), and a link library of object modules. These files are rt.db, rt.h, and rt.a, respectively. One set of files is shown there, but iconc in fact supports a search chain of these sets of files. The three files in each set share the same prefix. In the case of the standard run-time system, the prefix is rt. An alternate set of run-time routines might be named rt_a and be implemented with the files rt_a.db, rt_a.h, and rt_a.a.

The include file in the set is constructed by hand and includes the include file for the standard run-time library. If this is located in the directory /usr/icon/comp/src/runtime then rt_a.h contains

```
#include "/usr/icon/comp/src/runtime/rt.h"
```

rt_a.h must also include anything needed by the C compiler to compile the new run-time routines.

The data base and library are created and updated by rtt. The name of the data base is specified with rtt's –d option. The data base name must have a suffix of .db; rtt will supply the suffix if it is left off. The names of the other two files are constructed from the data base name. For example, operations in a file named my_fncs.rtt can be added to the alternate data base and library with the command

```
rtt –d rt_a my_fncs
```

The data base and library will be created if they do not exist.

The search chain for iconc is established through the environment variable DBLIST. To have iconc search the alternate data base and library before the standard ones use the command

```
setenv DBLIST rt_a
```

3. Making Existing C Libraries Available to Icon

One way to add functionality to the compiler's run-time system is to make C functions from existing libraries available to Icon. If the arguments and result types of these C functions consist only of integers, doubles (or floats), and strings, then adding them to the compiler's run-time system is simple. A number of the Icon functions in the standard run-time system are created from corresponding C functions. Several of these are used as examples in this section.

Ideally, the code produced by iconc to call an existing C function is simply a C call. However, it is usually necessary to convert values between Icon representations and C representations. Given this requirement, the best code produced by iconc performs three steps:

- Convert arguments from Icon representations to C representations.
- Call the C function with the converted arguments.
- Convert the function result value from a C representation to an Icon representation.

For built-in functions modeled after the examples in this section, iconc is usually able to generate such code.

Four parts of the implementation language are used when adding a C function to the Icon run-time system: a function header, type conversions, an abstract type computation, and in-line C code. Consider the existing built-in function sin. It simply makes the C sin function available to Icon.

A function header introduces the definition of the built-in function. The function header starts with function. This is followed by an indication of the length of the function's possible result sequences. The sin function produces exactly one result. A result sequence of length one is indicated by {1}.

As explained below, the Icon version of sin terminates with an error message if the argument is of the wrong type. It might seem that error termination should be considered a result sequence of length zero. However, a zero-length result sequence is used only to indicate failure (in the Icon sense). Possible error termination must be ignored when determining the length of a result sequence.

The next item in the header is the name of the Icon function. It may be the same as the C function or it may be different. The final item is a comma-separated list of parameters enclosed in parentheses. The only form of parameter needed here is a simple identifier. The header for the sin function is

```
function{1} sin(x)
```

The parameter x can be any Icon value. Even if it is of type real, it has a different representation than the corresponding C type, double. Type checking and conversion must be done before the C function is called with the value. The implementation language has several features that perform type checking and conversions. The conversion that is needed here has the form

cnv:*C-type*(*parameter*)

where *C-type* is C_integer, C_double, or C_string. C_integer indicates a C integer type (it may be long rather than int on 16 bit machines), C_double indicates the C double type, and C_string indicates a null-terminated character string. This from of cnv performs the standard Icon conversions between strings, integers, reals, and csets, but the result is a C value. The conversion establishes a new scope for the parameter's identifier. Within this scope the identifier refers to the C value. The conversion does not appear by itself; it is used in an if-then statement to check the success of the conversion. Typically, the success is checked with a negated conversion as it is in the conversion for sin:

```
if !cnv:C_double(x) then
   runerr(102, x)
```

runerr is an executable statement that results in a run-time error message. The message is determined by looking up the error number in a table (see the file data.c in the standard run-time system). If a value is supplied, it is printed. For example, if sin is called with a null value, the program terminates with the message

```
Run-time error 102
numeric expected
offending value: &null
```

Because runerr is on the execution path for the failure of the conversion, it sees the original parameter x, which is an Icon value.

Abstract type computations are used to inform iconc's type inferencing of the effects of the function. While these can be complicated, particularly for functions that update Icon data structures, for the functions addressed here, they are very simple. They indicate the type of value returned by the built-in function and are of the form

```
abstract {
   return icon-type
   }
```

Where the *icon-types* likely to be needed with pre-existing C functions are null, string, integer, and real. sin uses real.

The last feature of the implementation language needed for sin is in-line C code. It takes the form

```
inline {
   C code with extensions
   }
```

In this *extended C*, the C return statement is replaced by statements that perform an Icon-style return, suspend, or fail. Some forms of these statements also do C-to-Icon type conversions. For many C functions all that is needed is a return statement that does such a conversion. These return statments have the form

```
return C-type expr;
```

The *expr* must produce a C value of the indicated type. In this example, *expr* is simply the call to the C sin function:

```
return C_double sin(x);
```

Note that this use of x is in the scope of the type conversion and refers to a C value.

Putting these pieces together and preceding them by an optional comment in the form of a string literal results in the declaration

```
"sin(x), x in radians."
function{1} sin(x)
   if !cnv:C_double(x) then
     runerr(102, x)
   abstract {
      return real
      }
   inline {
      return C_double sin(x);
      }
end
```

Sometimes a straightforward implementation of a C function as an Icon built-in function is rather un-Iconish. For example, the C getenv function returns NULL if the requested environment variable does not exist. It makes sense for an Icon version of the function to fail under these circumstances rather than return the null value. The actual implementation of the getenv function is shown below. This function can either produce a result or fail, so it has a maximum result sequence length of 1 and a minimum result sequence length of 0. This is represented by {0,1} in the function header. Note the use of the extented C feature fail in the inline code.

```
"getenv(s) – return contents of environment variable s."
function{0,1} getenv(s)
   if !cnv:C_string(s) then
      runerr(103,s)
   abstract {
      return string
      }
   inline {
      register char *p;
```

```
   if ((p = getenv(s)) != NULL)      /* get environment variable */
      return C_string p;
   else                              /* fail if not in environment */
      fail;
   }
```

Another useful form of type conversion is def. This is used for supplying a default value. It has the form

def:*C-type*(*parameter*, *value*)

Is is similar to cnv. However, if *parameter* is null, the conversion does not fail. Instead, *value* is used for the converted value. The following built-in function makes the Unix sleep function available to Icon. It uses a default interval of 1 second. This function uses a return statement with no type conversion. The return statement's expression must evaluate to an Icon value, which is represented in C as a *descriptor* (see [4]). nulldesc is descriptor that always contains the null value.

```
function{1} sleep(n)
   if !def:C_integer(n, 1) then
      runerr(101, n)
   abstract {
      return null
      }
   inline {
      sleep((unsigned)n);
      return nulldesc;
      }
end
```

It is always a good idea to cast a C_integer to the desired C integer type in a C function call as demonstrated here.

The C sleep function returns an integer. This could be used as the result of the Icon built-in function, but the following will not work:

```
inline {
   return sleep((unsigned)n);
   }
```

This is because of a coding convention required by iconc for operations. This convention is that expressions on return statements do not have side effects. If the Icon program ignores the result of the built-in function, iconc may discard the entire return expression, eliminating the side effect along with the result. An auxiliary variable must be used in cases like this.

The Icon built-in function atan provides an example of a more complex function. It calls either the C function atan or atan2 depending on whether a second argument is given. It uses a type check of the form

is:*icon-type*(*parameter*)

This type check performs no conversions. It simply checks to see if *parameter* is of the desired type. Like the type conversions, it is used in an if statement. The code for atan is

```
"atan(x,y) — x,y  in radians; if y is present, produces atan2(x,y)."
function{1} atan(x,y)
   if !cnv:C_double(x) then
      runerr(102, x)
   abstract {
      return real
      }
```

```
   if is:null(y) then
      inline {
         return C_double atan(x);
         }
   if !cnv:C_double(y) then
      runerr(102, y)
   inline {
      return C_double atan2(x,y);
      }
end
```

This code is rather large to put in line. iconc performs type inferencing to try to determine the types of operands. If it is successful, it may be able to completely eliminate the type checking and the in-line code is small. If it does not infer the types of the operands, instead of putting the code with all the type checking in line, it calls a version of the function that rtt puts in the run-time library. This version contains full type checking code and dynamically decides whether to call atan or atan2.

Manifest Constants

Libraries of C functions are often accompanied by include files containing manifest constants. These can be made available to Icon as keywords. The declaration of a keyword is similar to that of a function, except that it starts with keyword and does not include a parameter list. The following example uses the manifest constant M_PI from math.h on the Sun 4 to implement the keyword &pi.

```
keyword{1} pi
   abstract {
      return real
      }
   inline {
      return C_double M_PI;
      }
end
```

For string, integer, and real constants, there is a simpler form of keyword declaration. The body of the declaration consists of const followed by the literal constant. &pi can be implemented as

```
#include <math.h>

keyword{1} pi
   const M_PI
end
```

Note the include statement. In many cases it does not matter whether a macro is expanded by rtt or later by the C compiler. In this case it does matter. rtt requires a literal constant after const, not an identifier. An integer literal must be in decimal format, not hexadecimal or octal format. Any numeric literal must be unsigned. When a keyword declared using const is encountered in an Icon program, the keyword is effectively replaced by the literal constant. No interpretation is done on the literal beyond simple lexical analysis (see [3] for details). When a C manifest constant from an include file is used here, its value must also be valid as an Icon literal; most C literals meet this requirement. The coding of a keyword using inline, such as the first version of &pi, does not have these restrictions. The expression on the return statement is C code.

4. Writing Operations From Scratch

Sometimes new Icon functions are needed that have no counterparts in a C library, or existing C library functions are too low-level to make reasonable Icon built-in functions. In these cases, more substantial built-in functions must be written.

Some facilities are best implemented as generators. Even if there are C functions that provide such facilities, they are, of course, not generators. As explained in Section 3, extended C includes an Icon-style suspend statement. It is used to create generators. Like the Icon suspend expression, execution continues after the statement if the operation is resumed. The suspend statement comes in the same forms as the return statement. A result sequence of arbitrary length is indicated by {*} in the operation header. The following code implements the seq function.

```
"seq(from, by) - generate from, from+by, from+2*by, ... ."
function{*} seq(from, by)
   if !def:C_integer(from, 1) then
      runerr(101, from)
   if !def:C_integer(by, 1) then
      runerr(101, by)
   abstract {
      return integer
      }
   body {
      /*
       * Produce error if by is 0, i.e., an infinite sequence of from's.
       */
      if (by == 0)
         irunerr(211, by);

      /*
       * Suspend sequence, stopping when largest or smallest integer
       *  is reached.
       */
      while ((from <= MaxLong && by > 0) || (from >= MinLong && by < 0)) {
         suspend C_integer from;
         from += by;
         }
      fail;
      }
end
```

Note the use of body instead of inline. When it is used, iconc still tries to simplify the type checking and puts the resulting code in line if it is simple enough. The body code is not placed in line. rtt puts it in a C library function. In in-line code, the body statement is replaced by a call to that function. Whether inline or body is used is a value judgement made by the programmer who writes the operation. The judgement is based on the size of the code and the desired speed of the operation. Not only does inline code eliminate a function call, iconc can optimize suspension in inline code more than it in body code. The code here is small enough that it might be reasonable to change body to inline.

which?

Some operations are polymorphous and take different actions based on the type of an argument. The type_case statement of the implementation language can be used for this. It similar to a switch statement (or an Icon case expression), but selection is based on the type of an argument rather than a value. The built-in function type makes use of it:

```
"type(x) - return type of x as a string."
function{1} type(x)
   abstract {
      return string
      }
```

```
   type_case x of {
      string:          inline { return C_string "string"; }
      null:            inline { return C_string "null"; }
      integer:         inline { return C_string "integer"; }
      real:            inline { return C_string "real"; }
      cset:            inline { return C_string "cset"; }
      file:            inline { return C_string "file"; }
      procedure:       inline { return C_string "procedure"; }
      list:            inline { return C_string "list"; }
      table:           inline { return C_string "table"; }
      set:             inline { return C_string "set"; }
      record:          inline { return BlkLoc(x)->record.recdesc->proc.recname; }
      co_expression: inline { return C_string "co-expression"; }
      default:
         runerr(123,x);
      }
end
```

See [4] for an explanation of the record entry.

Icon's storage management system includes a garbage collector. This places constraints on other parts of the run-time system. When a garbage collection occurs, it must be able to locate all references to objects in the string and block regions, and to co-expressions. Operations must be careful to keep all such references *visible* when storage allocations (including mallocs on some systems) might occur. It is important to note that storage allocations may occur while an operation is suspended.

Extended C has a tended declaration that insures that a variable is visible to garbage collection. For example, a block pointer is tended by the declaration

```
tended union block *variable;
```

This declaration is used by the key function while generating the keys of a table. The details of this code that deal with data structure manipulations are beyond the scope of this tutorial. See [4] for an explanation of the code that loops through the table. See [3] for an explanation of the abstract type computation. The implementation of key function is

```
"key(t) - generate successive keys (entry values) from table t."
function{*} key(t)
   if !is:table(t) then
      runerr(124, t)

   abstract {
      return store[type(t).key]
      }

   inline {
      tended union block *ep;
      register int i;

      for (i = 0; i < TSlots; i++) {
         for (ep = BlkLoc(t)->table.buckets[i];ep != NULL;ep = ep->telem.clink)
            suspend ep->telem.tref;
         }
      fail;
      }
end
```

malloc and free may be used for storage management in built-in operations. There is no facility for adding new data types to the Icon compiler. Because of support for type inferencing, doing so is very difficult and should only

be attempted with the aid of an expert on the compiler. (You are likely to need a very convincing justification to elicit that aid; it really is hard!)

rtt files may contain definitions of ordinary C functions to use as support routines for operations. Several extended features may be used in these otherwise ordinary C functions. These features include tended declarations, and type checking and conversions.

5. Exercises

The following are some suggested exercises for becoming familiar with adding functions and keywords to the Icon compiler. Use the instructions in Section 2 to set up an alternate data base and library for the compiler and add the following operations to it.

1. Create a new keyword &e with the value $e = 2.718281828459045$.

2. Use the C hyperbolic functions sinh, cosh, and tanh to create corresponding Icon built-in functions.

3. Make a copy of &features (it is in the compiler source in the file src/runtime/keyword.rtt) and add the string local additions to the generated output. Because iconc searches the alternate data base first, it will use the updated version of &features rather than the standard one.

4. Create a new keyword &fibseq that generates Fibonacci numbers. Fibonacci numbers are explained in [1]; an iterative version of the Icon procedure fibseq, as suggested in exercise 13.10, can be used as a model for the keyword.

5. Create a function lines(s) that takes as input a string consisting of lines separated by new lines and generates the starting locations of the lines. Warning, if you use a character pointer into an allocated string, it must be tended to insure it remains valid across suspensions. An example of the tended block pointer is given in Section 4; the declaration for a tended character pointer has the form

   tended char **variable*;

   Look in [3] at the conversion cnv:tmp_string. Can it be used in this function?

6. Use an existing string-analysis function as a model and create a version of lines (see the previous exercise) that can be used in string scanning.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.
2. K. Walker, *Using the Icon Compiler*, The Univ. of Arizona Icon Project Document IPD157, 1991.
3. K. Walker, *An Implementation Language for Icon Run-Time Routines*, The Univ. of Arizona Icon Project Document IPD79, 1989.
4. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.