Version 9 of the Icon Compiler

Ralph E. Griswold

Department of Computer Science, The University of Arizona

1. Introduction

There are two forms of the implementation of Icon, an interpreter and a compiler. The interpreter gets a program into execution quickly and is recommended for program development, debugging, and most production situations. The compiler produces code that executes somewhat faster than interpreted code (a factor of 2 or 3 is typical), but the compiler requires a large amount of resources and is very slow in producing executable code. The compiler is recommended only for small programs where execution speed is the paramount concern.

In most respects, the Icon compiler is compatible with the Icon interpreter [1]. Most programs developed under the interpreter run under the compiler with no changes.

It takes considerably longer to compile an Icon program than it does to translate it for interpretation. In most cases, it is advisable to do program development using the interpreter and then produce a production version of the program with the compiler.

Since compiler support for some features of Icon makes it difficult to produce fast executable code, these features are not available unless the compiler is told they are needed. This information is imparted by options to the compiler.

The compiler performs a number of optimizations to improve run-time performance. These optimizations can be disabled.

The Icon compiler produces C code, which is then compiled and linked to produce an executable program. Programs produced by the compiler stand alone and do not require a separate run-time system like the interpreter.

Only the routines needed by a compiled program are included in its executable file. Consequently, the size of a compiled program depends to a considerable extent on the range of features it uses.

The compiler itself requires a significant amount of memory. It must read a data base into memory, maintain a representation of the entire program, and perform analyses. Type inference in particular may require a large amount of memory, depending on the nature of the program. Icon programs on the order of several thousand lines may require several megabytes of memory to compile. In case a program does not compile with type inference enabled due to memory limitations or compiles slowly due to excessive paging, it may compile satisfactorily with type inferencing disabled.

The C code produced by the compiler is relatively voluminous and may require a large amount of disk space. Disabling all optimizations typically doubles the size of the generated C code.

Although the Icon compiler has been tested extensively at the Icon Project, it has not yet been used by a large and diverse programming community. Further use will certainly uncover bugs. If a compiled program appears to malfunction, first be sure it works properly with the Icon interpreter. Then try enabling optional features. Finally, try disabling all optimizations. Report bugs to the Icon Project. Include enough information, including the program and data, so that the problem can be replicated.

The next section describes how to run programs with the Icon compiler. The last section describes differences in language features between the compiler and interpreter.

2. Running the Compiler

The Icon compiler is invoked with the command

iconc [*options*] *files* [–x [*args*]]

where *options* can be any of the following:

–c stop compilation after producing C code

–e *file* redirect standard error output to *file*
–f *s* enable features as indicated by the letters in *s*
- a - all, equivalent to delns
- d - enable debugging features, including the effect of –f n (see below)
- e - enable error conversion
- l - enable large-integer arithmetic
- n - produce code that keeps track of line numbers and file names in the source code
- s - enable full string invocation

–m preprocess each source file with the m4 preprocessor (UNIX only)
–n *s* disable specific optimizations. These are indicated by the letters in *s*:
- a - all, equivalent to cest
- c - control flow optimizations other than switch statement optimizations
- e - expand operations in-line when reasonable (keywords are always put in-line)
- s - optimize switch statements associated with operation invocations
- t - type inference

–o *file* use *file* as the name of the executable file
–p *arg* pass *arg* on to the C compiler used by iconc
–r *path* use the run-time system at *path*.
–s suppress informative messages from the compiler
–t enable debugging features as for –f d and initialize &trace to –1
–u issue warning messages for undeclared identifiers in the program
–v *i* set verbosity level of informative messages to *i*
–C *prg* have iconc use the C compiler given by *prg*

File Names

One or more source file names must be specified. If a name has no suffix, .icn is appended. Otherwise, the name must already have a suffix of .icn. The argument ‘‘–’’ indicates that the source should be read from standard input. If the –o option is not used, the name of the executable file is constructed from the name of the first source file; stdin is used for ‘‘–’’. The compiler produces a C program source file and a C include file as intermediate files. The names of these files are constructed from the name of the executable file. For example,

```
iconc prog.icn
```

creates prog.c and prog.h. These files subsequently are deleted unless the –c option is specified. In any event, any previous versions of prog.c and prog.h are overwritten.

Options

See Section 3 for a description of features enabled by the –f options. Use of these features may reduce the optimizations performed by the compiler.

The –p option passes its argument to the C compiler. For example,

```
iconc –p "–w"
```

might be used to suppress warning messages. –p options accumulate; –p "" clears the list of options, including the initial set of standard options.

The –r option allows specification of the location of the directory that contains files needed by iconc. This overrides the default location. The *path* argument must be terminated by a slash, as in

```
iconc –r /usr/icon/lib/
```

The –v option controls the level of detail of iconc’s informative messages. –v 0 is the same as –s. –v 1 is the default behavior. Presently –v 2 and –v 3 only affect messages during type inferencing and are useful for monitoring its progress on large programs. –v 2 causes a dot to be written for each iteration of type inferencing. –v 3 writes an indication of the amount of information found in each iteration of type inferencing.

The –C option can be used to override the default C compiler used by iconc. It is possible, although unlikely, that using a different C compiler than the one expected when iconc is built may produce code incompatibilities.

As noted earlier, the –n t option sometimes is needed when compiling very large programs. The other –n options are used mostly for debugging the compiler.

Disabling control-flow optimizations by –n c may result in unreachable C code and corresponding warning messages when it is compiled. Such messages may be annoying, but they can safely be ignored. Occasionally unreachable C code may occur even with control-flow optimizations. This may indicate unreachable Icon code.

Environment Variables

The compiler determines the definition of a run-time operation from an operation data base. It links any needed routines from the link library associated with the data base. The compiler supports a search chain of these data bases and libraries. When searching a chain of data bases, it uses the first definition for an operation that it encounters. The environment variable DBLIST contains a blank separated list of data bases to be searched before the standard one. (The standard data base can be changed with the –r option.) An alternate data base may be placed in the search chain to provide new operations or to override standard operations.

The environment variable LPATH is used to specify paths to search for linked Icon source files. See the information on linking files in Section 3.

Automatic Execution

The –x option causes the compiled program to be executed automatically. Arguments to the program can be specified after the –x option, which must appear last. For example,

```
iconc prog –x test.log test.out
```

compiles and runs prog.icn with the arguments test.log and test.out.

3. Differences in Language Features

The compiler is designed to be compatible with the interpreter for Version 8.10 of Icon. This goal cannot be completely realized because several features of the language make optimizations difficult. Those features that cause problems and are not considered essential to the language have either been removed or made available only through compiler options.

Debugging, string invocation, linking code from other Icon programs, and external functions are handled somewhat differently in the compiler from the way they are handled in the interpreter. These differences are discussed in the following sections.

Large Integers

The –f l option enables large integer support. This allows the use of large integer literals and allows arithmetic expressions to use and produce large integer values. When this option is used, iconc produces code that is somewhat less efficient for arithmetic on ordinary integer values. This option may only be used with a run-time system that supports large integers. On most platforms, large integer support is the default; if large integers are not supported, iconc issues a warning and ignores the option.

Debugging Features

Several features of Icon provide debugging facilities. These features may require the generation of additional or more complex code. They normally are disabled, but they can be enabled by the –f d compiler option. An attempt to use a debugging function or keyword in a program where the compiler has not generated code to support it generally results in run-time error 402, program not compiled with debugging option.

The debugging facility in the compiler includes the printing of file names and line numbers with error messages and the printing of a procedure call trace-back when an error occurs. Even when a program is compiled with full debugging features enabled, there is no symbolic representation of the offending expression at the end of the traceback. The other operations falling under the heading of debugging features are:

```
display()
name()
variable()
&level
&trace
&line
&file
```

The –f n option enables limited debugging features. These are &line, &file, and the printing of line numbers and file names in error messages.

String Invocation

In the presence of string invocation, the compiler makes no attempt to infer what might be invoked. To fully support string invocation in the absence of other information, the compiler must link run-time routines for every operation. This results in a large executable file. String invocation is very useful in some programs, but it is not a heavily used feature. For these reasons, the compiler only supports string invocation to the extent explicitly requested by the programmer.

There are two ways for the programmer to enable support for string invocation. The –f s option makes all operations available for string invocation. Alternately, more selective control is possible using a new language feature, the invocable declaration. The syntax for this declaration is

invocable-declaration ::= invocable *operation {*, *operation }**

operation ::= string-literal |
string-literal : *integer-literal* |
all

Each *string-literal* is the string representation of an operation that might be invoked with string invocation. This may be a procedure, a record constructor, a built-in function, or an operator. For operators, the *integer-literal* indicates the number of arguments for the operator. If it is missing, all operators using that operator symbol (typically unary and binary versions) are made available. all indicates that all operations must be available for string invocation. For example, the following declaration indicates that binary +, binary and unary –, and write may be invoked with string invocation.

```
invocable "+":2, "–", "write"
```

There are some minor differences between string invocation in the compiler and in the interpreter. In the compiler, arguments to built-in operations are not automatically dereferenced, while they are in the interpreter. This means that string invocation on operations that make use of variables can produce different results in the two systems. For example, string invocation on assignment performs the assignment in the compiler, but produces a run-time error in the interpreter.

There are three additions to string invocation in the compiler. The string [...] represents list creation; it takes a variable number of arguments. The compiler treats & as an ordinary operator, so it is available for string invocation. In both the interpreter and the compiler, the string ... represents the ternary operation to-by. The interpreter implements to as a short-hand notation for to-by with a third argument of 1, so string invocation is not available for binary to. The compiler, on the other hand, implements to as a separate operation, and ... can be used in string invocation on two arguments.

If string invocation is attempted on an operation and there is no request to make it available, invocation may be unable to locate the operation and issue run-time error 106, procedure or integer expected. Because string invocation is under programmer control rather than being an Icon configuration option, it is no longer listed in &features.

The preceding discussion of string invocation also applies to the use of the built-in function proc.

External Functions

A search list of data bases/libraries combined with the features of the compiler's run-time implementation language [2] make it easy to call many C functions from within Icon. All that is required is a built-in function that converts between Icon and C values, and calls the C function. The call is typically specified as in-line code. The built-in function may be given the same name as the C function. The function is then added to a data base/library in the search list. The external function mechanism of the interpreter is not supported.

Linking Source Files

The compiler supports the link declaration, but it works differently than it does in the interpreter where it causes ucode files to be linked into the icode file. There are no ucode files in the compiler; instead the link declaration works on source files. It takes the file names from the link declaration, adds .icn suffixes, and compiles the files as if they had been specified on the command line. The source files are located using paths specified with the LPATH environment variable; this is analogous to the use of the IPATH environment variable in the interpreter.

Error Conversion

The –f e option allows the use of &error to enable the conversion of run-time errors into failure. Unless –f e is specified, the value of &error has no effect, and run-time errors are not converted to failure.

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, *A Tutorial on Adding Functions to the Icon Run-Time System*, The Univ. of Arizona Icon Project Document IPD173, 1991.