Transporting Version 6 of Icon*

Ralph E. Griswold

TR 86-25d

December 16, 1986; Last revised July 14, 1987

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

Transporting Version 6 of Icon

1. Background

The implementation of Version 6 of the Icon programming language is large and complex [1]. It is, however, written almost entirely in C, and it is designed to be portable to a wide range of computers and operating systems.

The implementation was developed on a UNIX* system. It has been installed on a wide range of UNIX systems, from mainframes to personal computers. Putting Icon on a new UNIX system is more a matter of installation than porting [2]. There presently also are installations of Icon for VAX/VMS and MS-DOS. This document addresses the problems and procedures for porting Icon to other operating systems and computers.

All installations of Version 6 of Icon are obtained from common source code, using conditional compilation to select system-dependent code. Consequently, transporting Icon to a new system is largely a matter of selecting appropriate values for configuration parameters, deciding among alternative definitions, and possibly adding some code that is computer- or operating-system-dependent.

A small amount of assembly-language code is needed for a complete installation. See Section 7. This code is optional and only affects co-expressions and checking for arithmetic overflow. A running version of the language can be obtained by working only in C.

Transporting Icon to a new system is a fairly complex task, although there are many aids to simplify the mechanical aspects. Read this report carefully before beginning a port. Understanding the Icon programming language is helpful during the debugging phase of a port. See [3-6].

2. Requirements

The C Compiler

The main requirement for implementing Icon is a production-quality C compiler that supports the *de facto* "K&R" standard [7]. The term "production quality" implies robustness, correctness, the ability to handle large files and complicated expressions, and a comprehensive run-time library.

C preprocessor should conform either to the ANSI C standard [8] or to the *de facto* standard for UNIX C preprocessors. In particular, Icon uses the C preprocessor to concatenate strings and substitute arguments within quotation marks. For the ANSI standard, the following definitions are used:

```
#define Cat(x,y) x##y
#define Lit(x) #x
```

For the UNIX *de facto* standard, the following definitions are used:

```
#define Ident(x) x
#define Cat(x,y) Ident(x)y
#define Lit(x) "x"
```

The following program can be used to test these preprocessor facilities:

*UNIX is a trademark of AT&T Bell Laboratories.

```
Cat(ma,in)()
{
   printf(Lit(Hello world\n));
}
```

If this program does not compile and print Hello world using one of the sets of definitions above, there is no point in proceeding. Contact the Icon Project as described in Section 8 for alternative approaches.

Memory

The Icon programming language requires a substantial amount of memory. While it will run on a computer with less than 256kb of user address space, its usefulness in such an environment is limited.

File Space

The source code for Icon is large — nearly 700kb. Compilation and testing require considerably more space. While the implementation can be divided into components that can be transported separately, this approach may be painful.

3. Organization of the Implementation

Icon was developed on a hierarchical file system. To facilitate file transfer between different operating systems and to simplify porting to systems that do not support file hierarchies, the source code for Icon is provided both in hierarchical form and in a "flat" form in which all files reside in the same area. This document applies to both the hierarchical and flat forms. Some of the supplementary documentation on Icon refers to file hierarchies. In interpreting this documentation on flat systems, simply ignore the directories in path specifications; the file names themselves are the same in the hierarchical and flat version.

3.1 Source Code

There are four components of Icon:

itran	a translator that converts source-language programs to *ucode*, an assembly language for an abstract "Icon machine".
ilink	a linker that combines one or more ucode files into a single binary *icode* file in executable format for the Icon machine.
iconx	an executor for icode, including a run-time system that supports the operations of the Icon language.
icont	a command processor that provides a user interface for running itran, ilink, and iconx.

itran

The translator is relatively straightforward. It contains a lexical analyzer, a parser, a code generator, and support routines. The translator is independent of the architecture of the computer on which it runs and its ucode output consists of printable text.

ilink

The linker is somewhat more complex than the translator. It reads ucode files and outputs binary code and data structures that are needed during execution. Because these data structures depend to some extent on the architecture of the computer on which Icon runs, the linker contains some machine-dependent parameters.

iconx

The executor is the largest and most complex of the components of Icon. Its data structures depend on the architecture of the computer on which Icon runs, and it includes code for all the operations in the Icon language. In addition, it manages storage dynamically.

icont

The command processor is a small program that executes the other components of Icon. Its main function is to analyze the user's command line and take appropriate actions. The command processor is a convenience, not a necessity. A command-language script can be used in its place.

The files related to the source are packaged in five groups:

itran	files for itran
ilink	files for ilink
iconx	files for iconx
icont	files for icont
h	header files

The header files are in a separate package, since some are used in several components of Icon. In some forms of the distribution, iconx comes in two parts, since it would not fit on a diskette otherwise.

Appendix A lists the files of each component of Icon. Some header files are used in several components; these are identified in the appendix. The files itran.bat, ilink.bat, iconx.bat, and icont.bat are scripts that indicate what files are to be compiled and loaded to produce the respective components. These scripts were derived from a UNIX implementation, but they can be adapted easily to other systems.

3.2 Tests

Test programs are divided into two parts. The first part, referred to as suite1, contains test programs and the expected output for itran and ilink. The second part, referred to as suite2, contains test programs and expected output for iconx.

See Section 6 for more information about the test programs.

4. An Overview of the Porting Process

The first step in the porting process is to configure the source code for the new system. This process is described in Section 5.1. After this is done, the components need to be ported, one by one.

The porting process for each component of Icon is essentially the same:

- provide code and definitions that are system-dependent
- compile the source files and link them to produce executable binary files
- test the result
- debug, iterating over the previous steps as necessary

The components should be ported in the order given above: itran, ilink, iconx, and icont. Of course, bugs in previously ported components may not show up until subsequent components are tested.

In addition to this obvious sequence of steps, some aspects of the implementation may be deferred until the entire system is running, or they may be implemented in a preliminary manner and subsequently refined. For example, the assembly-language portions of Icon are best left unimplemented until the rest of the system is running. Considerable frustration can be avoided if problems that come up can be circumvented with temporary expedients until the majority of the implementation is working properly. Similarly, conservative choices should be made during the initial phases of the implementation.

5. Conditional Compilation

Conditional compilation is used extensively in Icon to select code that is appropriate to a particular installation. Conceptually, conditional compilation can be divided into two categories: (1) matters related to the details of computer architecture, run-time system idiosyncrasies, specific C compilers, and operating-system variants. (2) matters that are specific to operating systems that are distinctly different, such as MS-DOS, UNIX, and VMS.

5.1 Parameters and Definitions

Most matters related to computer architecture, run-time systems, C compilers, and so forth are handled by defined constants, which should be taken care of before attempting to port the first component. Most of these constants are contained in config.h, which is included at the beginning of all files containing C source code. As distributed, this file contains definitions for a "vanilla" 32-bit computer. Conservative choices have been made for definitions where there is a choice. The portion of config.h in which changes may be needed is clearly marked.

ANSI Standard C

If your compiler supports the ANSI standard for C, add

```
#define Standard
```

to config.h. This will select the appropriate preprocessor definitions as described in Section 2.

Memory Management

Icon allocates memory dynamically and reclaims unused storage as necessary. Memory management is one of the most complicated portions of Icon, and it is the area of the implementation in which problems are most likely to occur.

Memory management is designed to work in a large portion of the user's data space within which Icon maintains regions for different kinds of data. Icon relocates data within these regions and expands the user's data space if more room is needed in a region. This expansion is done using the run-time library routines brk and sbrk.

This method of memory management is flexible and allows an Icon program to use available memory in the best way. However, expansion of the user's data space is not supported by all C compilers and operating systems. Furthermore, some C compilers do not provide complete support for brk and sbrk.

To overcome these problems, an alternative form of memory management is available using fixed-sized regions. One disadvantage of this form of memory management is that Icon may not be able to use all available memory. Nonetheless, this form of memory management is recommended for the initial phases of a port, since it avoids many potential problems that otherwise may complicate the porting process.

Memory management with fixed-sized regions is obtained by

```
#define FixedRegions
```

in config.h. This definition is included in the distributed file and must be removed if expandable regions are to be supported.

If fixed-sized regions are used, check memsize.h to see if the default sizes for the regions are suitable. It may be necessary to decrease these values for systems with a small amount of memory.

Data Sizing

The three symbols IntSize, LongSize, and PtrSize must be defined to be the sizes, in bits, of C ints, longs, and pointers. As mentioned in Section 2, only 16- and 32-bit sizes are appropriate; other sizes may lead to a variety of difficult problems.

Some C compilers provide several models in which these sizes are different. Since Icon requires a substantial amount of data space, if there is an option, chose a model that allows addressing of a large amount of memory. For example, in MS-DOS, Icon has been implemented for both the small-memory model (16-bit pointers) and the large-memory model (32-bit pointers). The small-memory-model implementation, however, cannot handle Icon programs that are large or need substantial amounts of run-time storage.

Character Set

The implementation of Icon was designed on the assumption it would run on computers using the ASCII character set. Recently work has been done toward supporting the EBCDIC character set as an alternative. If you are porting Icon to a computer that uses the EBCDIC character set, add

```
#define EBCDIC
```

to config.h.

Support for the EBCDIC character set is not yet complete and some places in the source code need modification for it. Examine code under #ifdef EBCDIC in fstr.c, fstranl.c, imain.c, lex.c, rconv.c, and toktab.c.

There also is an option that allows alternative representations of some characters commonly used in Icon programs that are not supported by many EBCDIC terminals and printers. The standard characters and their alternative forms are:

standard	*alternative*
{	$(
}	$)
[	$<
]	$>

To enable this option, add

```
#define ExtChars
```

to config.h.

Environmental and Architectural Considerations

If you are running in a 50-hz environment, change the definition of Hz in config.h from 60 to 50.

The definition

```
#define ZeroDivide
```

enables explicit handling of division by zero, as opposed to allowing it to be handled by a trap. This is a conservative choice and is provided by config.h as it is distributed. It can be removed if your system can handle division by zero via a trap.

Some systems require that C doubles be aligned at double-word boundaries. This is provided by

```
#define Double
```

This is a conservative choice and is provided in config.h as distributed.

Most computers have down-growing C stacks, for which stack addresses decrease as values are pushed. If your computer has an up-growing stack, for which stack addresses increase as values are pushed, add

```
#define UpStack
```

to config.h. *Note:* This definition only affects co-expressions.

Run-Time Library Considerations

C run-time libraries vary considerably in the facilities they provide. Icon provides some alternatives.

If your system does not support environment variables (via the run-time library routine getenv), add the following line to config.h:

```
#define NoEnvVars
```

This disables Icon's ability to change internal parameters to accommodate special user needs (such as using memory region sizes different from the defaults), but does not otherwise interfere with the use of Icon.

If your run-time library does not include qsort, add

```
#define IconQsort
```

to config.h. This causes a version of qsort in rmisc.c to be used. This should only be done if necessary, since a version of qsort in a system library is likely to be more efficient than the one in rmisc.c.

If your run-time library does not include gcvt or if it does not produce acceptable results (try write(0.3) once you have Icon running), add

```
#define IconGcvt
```

to config.h. This causes a version of gcvt in rmisc.c to be used.

If your run-time library does not include memset or memcpy, add

```
#define IconMem
```

to config.h. This causes a version of these routines in rmisc.c to be used.

The routine atof is used in the Icon linker to convert strings for real literals to corresponding floating-point numbers. If the version of atof on your system does not work properly, add

```
#define NoAtof
```

which replaces the use of atof by in-line conversion code.

Different C run-time libraries use different names for routines for locating substrings within strings. The Icon source code uses index and rindex. The other possible names for the routines are strchr and strrchr. Since strchr and strrchr are more common than index and rindex, the following definitions are included in config.h as distributed:

```
#define index strchr
#define rindex strrchr
```

Remove them if your system uses index and rindex.

Optional Features

Co-expression activation requires a simple context switch, which must be written in assembly language. In config.h, there is

```
#define NoCoexpr
```

This does not disable all co-expression facilities, but it omits code related to co-expressions that otherwise would be compiled. If the co-expression context switch is implemented as described in Section 7.1, this definition should be deleted.

Since C does not provide checking for arithmetic overflow, this feature of Icon must be written in assembly language. In config.h as distributed, there is

```
#define NoOver
```

This disables arithmetic overflow checking. If arithmetic overflow checking is implemented as described in Section 7.2, this definition should be deleted.

Local Information

The value of HostStr is used in the Icon keyword &host. Change the value is config.h as distributed to a value appropriate to your system.

The locations where the various components of Icon are expected to reside are used in several places, although their actual values are not important unless icont is implemented. These values are defined in path.h. As distributed, path.h contains values appropriate for MS-DOS. These values should be checked and changed if icont is implemented.

5.2 Operating System Differences

Conditional compilation because of operating system differences usually is due to differences in file naming, the handling of input and output, and environmental factors.

The presently supported operating system are AmigaDos, Atari ST TOS, MPW on the Macintosh, MS-DOS, UNIX, and VMS. There is a hook for transporting to another unspecified operating system (PORT).

The way conditional compilation is handled for matters specific to different operating systems relies on a distinct name being associated with each operating system and there being a definition for *all* operating system names. The definition of a name is 1 if compilation is being done for that system, but 0 otherwise. For example, when Icon is being compiled to run on a UNIX system, the names are defined as follows:

```
#define PORT 0
#define AMIGA 0
#define ATARI_ST 0
#define MACINTOSH 0
#define MSDOS 0
#define MVS 0
#define UNIX 1
#define VM 0
#define VMS 0
```

These definitions occur in config.h, a header file that is included in all files that contain C code.

These names are used in logical conditionals, such as

```
#if UNIX || VMS
        .
        .           /* code for UNIX and VMS systems */
        .
#endif
#if MSDOS
        .
        .           /* code for MS-DOS */
        .
#endif
```

Logical conditionals with #if are used instead of defined or undefined names with #ifdef to avoid nested conditionals, which become very complicated and difficult to understand when there are several alternative operating systems. This method does, however, require that all operating system names be defined and that all but one have the value 0. It also is important not to use #ifdef accidentally in place of #if, since all the names are defined.

To set things up for a port to a new operating system, define the new name to be 1 and change all others as 0. For example, It may be convenient to use PORT when starting and change it to the appropriate new name later:

```
#define PORT 1
#define AMIGA 0
#define ATARI_ST 0
#define MACINTOSH 0
#define MSDOS 0
#define MVS 0
#define UNIX 0
#define VM 0
#define VMS 0
```

When Icon is transported to a new operating system, it is necessary to locate all the places where there is conditional compilation that is operating-system dependent. To make this easy, such code is bracketed by unique comments of the following form:

```
/*
 * The following code is operating-system dependent.
 */
                        .
                        .
                        .
/*
 * End of operating-system specific code.
 */
```

The files that contain operating-system-dependent code are listed in Appendix B. There presently are 77 segments in all that contain such code. Each segment contains comments that describe the purpose of the code. In most cases, the most likely code or a suggestion is given in the conditional code under PORT. In some cases, no code will be needed. In others, code for an existing system may suffice for the new system.

In any event, code for the new operating system name must be added to each such segment, either by adding it to a logical disjunction, as in

```
#if MSDOS || UNIX || PORT
        .
        .
        .
#endif

#if VMS
        .
        .
        .
#endif
```

or by filling in the segment with the appropriate code, as in

```
#if PORT
            .
            .           /* code for the the port */
            .
#endif
```

If the code for the new operating system is empty, a comment should be added so that it is clear the dependency has been considered.

If an operating-system dependency is encountered at a place where it did not exist previously, a new section with the bracketing comments for all operating systems must be set up appropriately. Do not simply add code like

```
#if PORT
        .
        .
        .
#endif
```

without empty code for the other systems, since this will interfere with transportation to other systems in the future.

Do not use #else constructions; this encourages errors and obscures the mutually exclusive nature of operating system differences.

6. Building and Testing

6.1 The Translator

Start by compiling all the C programs listed in tran.bat. Load the resulting object files to produce itran. With any luck, this will go without problems, since itran is largely machine-independent. If you encounter problems, check the portions of code containing operating system dependencies.

Once you have a version of itran, try it on the Icon programs in suite1. For example, to translate bitops.icn in suite1, do

```
itran bitops.icn
```

This should yield two ucode files, bitops.u1 and bitops.u2. The .u1 file contains procedure declarations and code for the Icon machine; the .u2 file contains global declaration information.

These files both consist of printable text. They should be identical to the corresponding files in suite1. Be careful to run itran in a way that does not overwrite the ucode files in suite1.

More than likely, if you get any ucode files at all, they will be correct, since the translator is machine-independent and portable and no significant problems have been encountered with it in other ports.

6.2 The Linker

Compile all the C files listed in ilink.bat and load them to get ilink. You may encounter more problems here, since the linker is somewhat dependent on the sizes of C data objects and does more sophisticated input and output than the translator.

Once you have ilink, test it on ucode files as follows:

```
ilink -D bitops.u1
```

Only the .u1 file is named for the linker, but both the .u1 and .u2 files must be present for ilink. The -D option causes ilink to produce a debugging output file with the suffix .ux — in this case, bitops.ux. Compare this file to the corresponding one in suite1. The two files may not be exactly the same, since the debugging output depends on C sizes (the distributed files are for 32-bit ints) and idiosyncrasies of fprintf. However, the files should be similar. Do not worry about small differences.

6.3 The Run-Time System

If you get this far without apparent problems, you are ready for the next — and most difficult — part of the transporting process: iconx. Compile all the C programs listed in iconx.bat and load them to form iconx.

As a first test, try iconx on hello.icn in suite1 as follows:

```
itran hello.icn
ilink hello.u1
iconx hello
```

If all is well, the last step should print out "hello world" and some identifying information.

Once this test has been passed, more rigorous testing should follow. At this point, you probably will want to devise a way of testing programs, since there are a large number of tests. This is done for the MS-DOS implementation using the following .bat file:

```
echo off
itran -s %1.icn
ilink %1.u1
echo Executing
if not exist %1.dat goto skip1
iconx  %1 <%1.dat >%1.out
goto comp
:skip1
iconx -e - %1 >%1.out
:comp
compare %1.out stand\%1.out -t -w -l %1.dif
type %1.dif
del %1
del %1.u1
del %1.u2
erase local\%1.out
erase local\%1.dif
movefile %1.out local
movefile %1.dif local
echo on
```

On UNIX systems, the following script is used to test all the files in a list:

```
for i in 'cat $1.lst'
do
   rm -f local/$i.out
   echo Running $i
   itran $i.icn
   ilink $i.u1
   if test -r $i.dat
   then
      iconx $i <$i.dat >local/$i.out 2>&1
   else
      iconx $i >local/$i.out 2>&1
   fi
   echo Checking $i
   diff local/$i.out stand/$i.out
   rm -f $i
done
```

While these methods are adapted to specific operating-system capabilities, something similar can be concocted for other systems. Making such a facility as easy to use as possible is worth the effort.

In suite2 there many Icon programs for testing different aspects of iconx. These range from simple tests to "grinders". The names of the test programs are listed in the following files:

icon.lst	short programs*
expr.lst	programs that contain a wide variety of expressions
check.lst	programs that may produce different results on different systems
extra.lst	programs that test additional features
work.lst	long-running programs*

The lists flagged with a * contains tests that require data files that are included in suite2 with the ending .dat. For example, the Icon program meander.icn, listed in icon.lst, takes data from meander.dat. suite2 also contains files whose names end in .out that contain the expected output of each test program. For example, the expected output of meander.icn is contained in meander.out.

The programs listed in icon.lst should produce output identical to that in their corresponding .out files. Any

discrepancies should be checked carefully and corrections made before continuing.

The programs listed in expr.lst execute a wide variety of individual expressions. Ideally, there should be no discrepancies between their output and the expected output. If there are many discrepancies, something serious probably is wrong. If there are only a few discrepancies, they may be noted while other testing is conducted.

The programs listed in check.lst certainly will show some differences, since they test features whose results are time- and environment-dependent. Other differences may show up also. These do not necessarily indicate problems. For examples, minor differences in the results of floating-point arithmetic are common in these tests.

The programs listed in extra.lst test some features that are not tested elsewhere. They should be treated like the programs listed in icon.lst.

In work.lst are listed several programs that run for a very long time and exercise some parts of the implementation extensively. When running these programs, keep in mind that they are expected to run for a considerable amount of time. For example, they take a total of about 32 minutes of wall-clock time on an IBM XT running LMM MS-DOS Icon.

Since storage management is one of the parts of Icon that is likely to give trouble, there are special storage-management tests:

lgc.lst	for systems that have 32-bit pointers
sgc.lst	for systems that have 16-bit pointers

One program shows a difference in output if the fixed-regions version of memory management is used, since it runs out of space.

These programs run for a long period of time. For example, the programs in lgc.lst take a total of about 18 minutes of wall-clock time on an IBM XT running LMM MS-DOS Icon.

Not much general advice can be given about locating and correcting problems that may show up in testing iconx. It has to be done the hard way and may involve learning more about the Icon language [3] and how it is implemented [1]. A good debugger can be very helpful.

6.4 The Command Processor

See the remarks about icont in Section 3.1. If icont is to be implemented, compile icont.c and load it to form icont.

Test icont on hello.icn in suite1 as follows:

```
icont hello.icn -x
```

If all is well, hello.icn should be translated, linked, and executed to produce the output given in the preceding section.

icont supports several options. See [6] for a complete list. Various combinations of options should be tested to confirm that icont is working properly.

7. Assembly-Language Code

Once Icon is running satisfactorily, you may wish to implement the features that require assembly language: co-expressions and arithmetic overflow checking.

7.1 Co-Expressions

All aspects of co-expression creation and activation are written in C in Version 6 except for a routine, coswitch, that is needed for context switching. This routine requires assembly language, since it must manipulate hardware registers. It either can be written as a C routine with asm directives or as an assembly language routine.

The file rswitch.c, as distributed, contains a version of coswitch that results in error termination if an Icon program attempts to activate a co-expression. This file needs to be modified or replaced by one that contains a real coswitch routine.

Calls to the context switch have the form coswitch(old_cs,new_cs,first), where old_cs is a pointer to an array

of words that contain C state information for the current co-expression, new_cs is a pointer to an array of words that hold C state information for a co-expression to be activated, and first is 1 or 0, depending on whether or not the new co-expression has or has not been activated before. The zeroth element of a C state array always contains the hardware stack pointer (*sp*) for that co-expression. The other elements can be used to save any C frame pointers and any other registers your C compiler expects to be preserved across calls.

The default number of elements for saving the C state is 15. This number may be changed by adding

```
#define CStateSize n
```

to config.h, where *n* is the number of elements needed.

The first thing coswitch does is to save the current pointers and registers in the old_cs array. Then it tests first. If first is zero, coswitch sets *sp* from new_cs[0], clears the C frame pointers, and *calls* interp. If first is not zero, it loads the (previously saved) *sp*, C frame pointers, and registers from new_cs and returns.

Written in C, coswitch has the form:

```
/*
 * coswitch
 */
coswitch(old_cs, new_cs, first)
int *old_cs, *new_cs;
int first;
{
                                    .
                                    .
                                    .
            /* save sp, frame pointers, and other registers in old_cs */
                                    .
                                    .
                                    .
   if (first == 0) {/* this is first activation */
                                    .
                                    .
                                    .
            /* load sp from new_cs[0] and clear frame pointers */
                                    .
                                    .
                                    .
      interp(0, 0);
      syserr("interp() returned in coswitch");
      }

   else {
                                    .
                                    .
                                    .
            /* load sp, frame pointers, and other registers from new_cs */
                                    .
                                    .
                                    .
      }
}
```

Two sample co-expression context switches are included with the source files: rswitch.dos for MS-DOS and rswitch.sun for the Sun Workstation (which has a Motorola 68000 processor).

When coswitch is implemented, remove the definition for NoCoexpr in config.h and rebuild iconx.

There are two lists of Icon programs for testing co-expressions:

lcoexpr.lst	for systems with 32-bit pointers
scoexpr.lst	for systems with 16-bit pointers

For systems with small address spaces, some of the tests may terminate prematurely because of lack of memory.

If you have trouble with your context switch, the first thing to do is double-check the registers that your C compiler expects to be preserved across calls — different C compilers on the same computer may have different requirements.

Another possible source of problems is built-in stack checking. Co-expressions rely on being able to specify an arbitrary region of memory for the C stack. If your C compiler generates code for stack probes that expects the C stack to be at a specific location, you may need to disable this code or replace it with something more appropriate.

If your system does not allow the C stack to be at an arbitrary place in memory, there is probably little hope of implementing co-expressions.

7.2 Arithmetic Overflow Checks

C does not provide overflow checking for integer addition, subtraction, or multiplication. Icon, on the other hand, is supposed to check for overflow. This usually requires assembly-language code.

If you do not want to implement overflow checking, you need do nothing. If you want to implement overflow checking, remove the definition of NoOver from config.h and write routines ckadd, cksub, and ckmul that call runerr(203,0) in the case of overflow. Note that testing multiplicative overflow may be difficult on some computers and this check need not be implemented even though the others are.

Two sample routines for checking overflow are included with the source files: rover.dos for MS-DOS and rover.sun for the Sun Workstation.

The file over.lst contains the names of programs to use for testing arithmetic overflow. You should also rerun previous tests at this point to make sure that arithmetic still works properly.

8. Trouble Reports and Feedback

If you run into problems, contact us at the Icon Project:

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

(602) 621-6613

icon-project@arizona.edu (Internet)
... {allegra, cmcl2, ihnp4, noao}!arizona!icon-project (uucp)

Please also let us know of any suggestions for improvements to the porting process.

Once you have completed your port, please send us copies of any files that you modified so that we can make corresponding changes in the central version of the source code. Once this is done, you can get a new copy of the source code whenever changes or extensions are made to the implementation.

Also be sure to include documentation on any features that are not implemented in your port or any changes that would affect users.

Acknowledgements

Many persons have been involved in the implementation of Icon. Contributions to its portability have been made by Bill Mitchell, Kelvin Nilsen, Gregg Townsend, and Cheyenne Wills.

Bill Mitchell and Janalee O'Bagy provided several useful suggestions for the presentation of the material in this report.

References

1. R. E. Griswold and M. T. Griswold, *The Implementation of The Icon Programming Language*, Princeton University Press, 1986.

2. R. E. Griswold, *Installation Guide for Version 6 of Icon on UNIX Systems*, The Univ. of Arizona Tech. Rep. 86-11d, 1986.

3. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

4. R. E. Griswold, *An Overview of the Icon Programming Language*, The Univ. of Arizona Tech. Rep. 83-3c, 1983, Revised 1986.

5. R. E. Griswold, W. H. Mitchell and J. O'Bagy, *Version 6 of Icon*, The Univ. of Arizona Tech. Rep. 86-10b, 1986.

6. R. E. Griswold, *Version 6 of Icon for MS-DOS*, The Univ. of Arizona Icon Project Document IPD2, 1987.

7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

8. Technical Committee X3J11, *Draft Proposed American National Standard for Information Systems — Programming Language C*, Oct. 1986.

Appendix A — Files Used by Components of Icon

Files marked by * are used in more than one component.

Files Used by itran

config.h*	general configuration information
keyword.h*	keyword definitions
itran.h	general heading information of itran
lex.h	header for lexical analysis
lfile.h	header for link declarations
sym.h	header for symbol tables
token.h	token definitions
tstats.h	statistics gathering definitions (normally not used)
tree.h	code tree information
version.h*	version information
code.c	code generator
err.c	error messages
itran.c	main program
keyword.c	keyword structure
lex.c	lexical analyzer
lnklist.c	file linking
mem.c	memory management
optab.c	state tables for operator recognition
parse.c	parser
sym.c	symbol table management
toktab.c	token table
tree.c	code tree constructor

Files Used by ilink

config.h*	general configuration information
cpuconf.h*	computer architecture information
fdefs.h*	function definitions
header.h*	icode header structure
ilink.h	general heading information for ilink
keyword.h*	keyword definitions
opcode.h	opcode structure
opdefs.h*	icode instruction definitions
paths.h*	file paths
rt.h*	header for run-time system
version.h*	version information
glob.c	processor for global information
ilink.c	main program
lcode.c	code generator
llex.c	lexical analyzer
lmem.c	memory management
lsym.c	symbol table management
opcode.c	opcode table

Files Used by iconx

config.h*	general configuration information
cpuconf.h*	computer configuration information
fdefs.h*	function definitions
gc.h	garbage collection definitions
header.h*	icode header
keyword.h*	keyword definitions
memsize.h*	memory sizing
opdefs.h*	icode definitions
rt.h*	run-time definitions
version.h*	version information
fconv.c	conversion functions
fmisc.c	miscellaneous functions
fscan.c	scanning functions
fstr.c	string construction functions
fstranl.c	string analysis functions
fstruct.c	data structure functions
fsys.c	system functions
fxtra.c	extra functions
idata.c	data
imain.c	main program
interp.c	icode interpreter
invoke.c	function and procedure invocation
lmisc.c	miscellaneous library routines
lrec.c	library routines for record
lscan.c	scanning routines
oarith.c	arithmetic operations
oasgn.c	assignment operations
ocat.c	concatenation operations
ocomp.c	comparison operations
omisc.c	miscellaneous operations
oref.c	referencing operations
oset.c	set operations
ovalue.c	value operations
rcomp.c	comparison routines
rconv.c	conversion routines
rdefault.c	default value routines
rdoasgn.c	assignment routines
rlocal.c	locally needed routines
rmemmgt.c	memory management routines
rmisc.c	miscellaneous routines
rover.c	arithmetic overflow routines
rstruct.c	structure routines
rswitch.c	co-expression context-switching routine
rsys.c	system routines

Files Used by icont

config.h*	general configuration information
paths.h*	path definitions
icont.c	main program

Appendix B — System-Dependent Code

The following source files contain code that is operating-system dependent. The number of places where such code occurs in each file is given in parentheses.

Translator:

itran.c (5)

Linker

ilink.c (7)
lcode.c (2)
llex.c (1)
lmem.c (6)

Executor:

fmisc.c (1)
fsys.c (12)
imain.c (9)
interp.c (4)
lmisc.c (6)
rconv.c (2)
rmemmgt.c (3)
rmisc.c (3)
rsys.c (3)

Command Processor

icont.c (10)