Transporting Version 7.5 of Icon*

Ralph E. Griswold

TR 88-9c

January 29, 1988; last revised November 23, 1988

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 7.5 of Icon

1. Background

The implementation 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 implementations of Icon for the Amiga, the Atari ST, the Macintosh under MPW, MS-DOS, OS/2, VM/CMS, and VMS. This document addresses the problems and procedures for porting Icon to other operating systems and computers.

The current version of Icon is 7.5 [3]. All installations of Version 7.5 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 portions. Read this report carefully before beginning a port. Understanding the Icon programming language is helpful during the debugging phase of a port. See [3-5]

2. Requirements

C Data Sizes

Icon places the following requirements on C data sizes:

- *chars* must be 8 bits.
- *ints* must be 16, 32, or 64 bits.
- *longs* and pointers must be 32 or 64 bits.
- All pointers must be the same length.
- *longs* and pointers must be the same length.

If your C data sizes do not meet these requirements, do not attempt to transport Icon. Call the Icon Project for advice. *Note:* Icon has not yet been ported to a computer with 64-bit C data sizes; such a port may encounter problems.

The C Compiler

The main requirement for implementing Icon is a production-quality C compiler that supports at least the *de facto* "K&R" standard [6]. 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 [7] 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:

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

```
#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:

```
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. The practical minimum is 512kb.

File Space

The source code for Icon is large — about 850kb. 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 for 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 two components of Icon:

- icont — a command processor that converts source-language programs into *icode*, the ''executable binary'' for the Icon virtual machine.
- iconx — an executor for icode, including a run-time system that supports the operations of the Icon language.

The files related to the source are packaged in three sections:

h	header files
icont	files for icont
iconx	files for iconx

The header files are in a separate package, since some are used in both components of Icon. In some forms of the diskette distribution, iconx comes in two parts, since it is is too large to fit on some kinds of diskettes.

Appendix A lists the files of each component of Icon. Some header files are used in both components; these are identified in the appendix. The files icont.bat and iconx.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 icont. 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, icont and iconx need to be constructed.

The process for each component 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

icont needs to be ported before iconx, since the output of icont is needed to test iconx. Of course, bugs in icont may not show up until iconx is 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 iconx 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

There are many defined constants in the source code for Icon that vary from system to system. Default values are provided so that the usual cases are handled automatically. The file define.h contains C preprocessor definitions for parameters that differ from the defaults or that must be provided on an individual basis. This file initially contains values for a ''vanilla'' 32-bit system. If your system closely approximates a ''vanilla'' system, you will have few changes to make to define.h. Over the range of possible systems, there are many possibilities as described below. *Do not be intimidated by the large number of options that follow;* only a few are needed for any implementation.

The definitions are grouped into categories so that any necessary changes to define.h can be approached in a logical way.

C compiler considerations: If your C compiler supports the ANSI C draft standard, add

```
#define Standard
```

to define.h.

This has several effects. One is to use the ANSI C mechanism for token concatenation and substitution in quotes during preprocessing. Another is to provide a typedef for pointer that is void * rather than char *. It also enables the use of the void type for functions that do not return values.

If your C compiler supports the void type but not the ANSI C draft standard, add

```
#define VoidType
```

to define.h.

If your C compiler supports function prototypes, add

```
#define Prototypes
```

to define.h. This causes function prototypes (in proto.h) to be used in place of forward declarations. The use of prototypes may be very helpful in getting Icon to work, especially on systems with 16-bit *ints* or unusual pointer representations. This option is not automatically enabled by the definition of Standard, since there are C compilers that support (or require) ANSI C constructions but which have trouble with prototypes.

On some systems it may be necessary to provide a different typedef for pointer than mentioned above. For example, on the huge-memory-model implementation of Icon for Microsoft C on MS-DOS, its define.h contains

```
typedef huge void *pointer
```

If an alternative typedef is used for pointer, add

```
#define PointerDef
```

to define.h to avoid the default one.

Sometimes there are problems with pointer arithmetic. There are three macros used for pointer arithmetic, with the default definitions

```
#define IncrPtr(p,i) ((p)+(word)(i))
#define DecrPtr(p,i) ((p)-(word)(i))
#define DiffPtrs(p1,p2) (word)((p1)-(p2))
```

word is a typedef that is provided automatically and usually is *long int.*

Overriding definitions can be provided in define.h. For example, the huge-memory-model implementation of Icon for MS-DOS uses

```
#define DiffPtrs(p1,p2) (word)((p1)-(p2))
```

If you provide alternate definitions for pointer arithmetic, be careful to enclose all arguments in parentheses. *Note:* The use of macros for pointer arithmetic is incomplete in the present version of the source code.

C sizing and alignment: There are four constants that relate to the size of C data and alignment:

IntBits	(default: 32)
WordBits	(default: 32)
Double	(default: undefined)

IntBits is the number of bits in a C *int*. It may be 16, 32, or 64. WordBits is the number of bits in a C *long* (Icon's "word"). It may be 32 or 64. If your C library expects *doubles* to be aligned at double-word boundaries, add

```
#define Double
```

to define.h.

Floating-point arithmetic: There are four optional definitions related to floating-point arithmetic:

Big	(default: 9007199254740092.)
LogHuge	(default: 309)
Precision	(default: 10)
ZeroDivide	(default: undefined)

The values of Big, LogHuge, and Precision give, respectively, the largest floating-point number that does not loose precision, the maximum base-10 exponent + 1 of a floating-point number, and the number of digits provided in the string representation of a floating-point number. If the default values given above do not suit the floating-point arithmetic on your system, add appropriate definitions to define.h. If your system needs a software check for division by floating-point zero, add

```
#define ZeroDivide
```

to define.h.

Include file location: The location of the include file time.h varies from system to system. Its default location is <time.h>. If it resides at a different location on your system (usually <sys/time.h>), add an appropriate definition of SysTime to define.h, as in

```
#define SysTime <sys/time.h>
```

If the location is incorrect, a fatal error will occur during the compilation of src/iconx/lmisc.c.

The use of this definition also depends on your C preprocessor making macro substitutions in #include directives. Most preprocessors do, but if yours does not, edit src/iconx/lmisc.c and replace SysTime there by the appropriate value. If you have to do this, make a note to come back later and place the definition under the control of conditional compilation as described in Step 4.

Run-time routines: The support for some run-time routines varies from system to system. The related constants are:

IconGcvt	(default: undefined)
IconQsort	(default: undefined)
NoAtof	(default: undefined)
SysMem	(default: undefined)
index	(default: undefined)
rindex	(default: undefined)

If IconGcvt and IconQsort are defined, versions of *gcvt()* and *qsort()* in the Icon system are used in place of the routines normally provided in the C run-time system. These constants only need to be defined if the versions of these routines in your run-time system are defective or missing.

The C run-time routine *atof()* normally 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
```

to define.h. This replaces the use of *atof* by in-line conversion code.

If your run-time system includes *memcpy()* and *memset()*, add

```
#define SysMem
```

to define.h. Otherwise, versions of these routines in the Icon system are used.

Different C compilers use different names for the routines for locating substrings within strings. The source code for Icon uses index and rindex. The other possibilities are strchr and strrchr. If your system uses the latter names, add

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

to define.h.

Storage management: Icon includes its own versions of *malloc()*, *calloc()*, *realloc()*, and *free()* so that it can manage its storage region without interference from allocation by the operating system. Normally, Icon's versions of these routines are loaded instead of the system library routines.

Leave things are they are in the initial configuration, but if your system insists on loading its own library routines, multiple definitions will occur as a result of the *ld* in src/iconx. If multiple definitions occur, go back and add

```
#define IconAlloc
```

to define.h. This definition causes Icon's routines to be named differently to avoid collision with the system routine names.

One possible effect of this definition is to interfere with Icon's expansion of its memory region in case the initial values for allocated storage are not large enough to accommodate a program that produces a lot of data. This problem appears in the form of run-time errors 305-307. Users can get around this problem on a case-by-case basis by increasing the initial values for allocated storage by setting environment variables [7].

Icon's dynamic storage allocation system uses three memory regions. In some implementations, these regions expand if necessary, allowing memory space to be used in a flexible fashion. This "expandable regions" method relies on the use of *brk()* and *sbrk()* and the system treatment of user memory space as one logically contiguous region. This method does not work on many systems that treat memory as segmented or do not support *brk* and *sbrk*. On such systems, fixed-sized regions are used. Since this is the commonest case,

```
#define FixedRegions
```

is included in define.h initially. If your system supports *brk()* and *sbrk()*, you may wish to remove this definition in order to get better utilization of memory. However, since expandable regions are more prone to problems than fixed regions, it is wise to start with the latter and try the former only after everything else is working.

Storage regions: The sizes of Icon's run-time storage regions for allocated blocks and strings normally are the same for all implementations. However, different values can be set:

MaxAbrSize (default: 65000)
MaxStrSize (default: 65000)

Since users can override the set values with environment variables, it is unwise to change them from their defaults except in unusual cases.

The sizes for Icon's main interpreter stack and co-expression stacks also can be set:

MStackSize (default: 10000)
StackSize (default: 2000)

As for the block and string storage regions, it is unwise to change the default values except in unusual cases.

Finally, with fixed-regions storage management, a list used for pointers to strings during garbage collection, can be sized:

QualLstSize (default: 5000)

Like the sizes above, this one normally is best left unchanged.

Allocation size: Normally *malloc()* is used to allocate space for Icon's storage regions. This limits region sizes to the value of the largest *unsigned int*. Some systems provide alternative allocation routines for allocating larger regions. To change the allocation procedure for regions, add a definition for AllocReg to define.h. For example, the huge-memory-model implementation of Icon for Microsoft C uses the following:

```
#define AllocReg(n) halloc((long)n,sizeof(char))
```

Note: Icon still uses *malloc()* for allocating other blocks. If this is a problem, it may be possible to change this by defining malloc in define.h, as in

```
#define malloc lmalloc
```

If this is done, and the size of the allocation is not *unsigned int*, add an appropriate definition for the type by defining AllocType in define.h, such as

```
#define AllocType unsigned long int
```

It is also necessary to add a definition for the limit on the size of an Icon region:

#define MaxBlock *n*

where *n* is the maximum size allowed (the default for MaxBlock is MaxUnsigned, the largest *unsigned int*). It generally is not advisable to set MaxBlock to the largest size an alternative allocation routine can return. For the huge-memory-model implementation mentioned above, MaxBlock is 256000.

File name suffixes: The suffixes used to identify Icon source programs, ucode files, and icode files may be specified in define.h:

```
#define SourceSuffix     (default: ".icn")
#define U1Suffix         (default: ".u1")
#define U2Suffix         (default: ".u2")
#define USuffix          (default: ".u")
#define IcodeSuffix      (default: "")
#define IcodeASuffix     (default: "")
```

USuffix is used for the abbreviation that icont understands in place of the complete U1Suffix or U2Suffix. IcodeASuffix is an alternative suffix that iconx uses when searching for icode files specified without a suffix. For example, on MS-DOS, IcodeSuffix is ".icx" and IcodeASuffix is ".ICX".

If values other than the defaults are specified, care must be taken not to introduce conflicts or collisions among names of different types of files.

Paths: If icont is given a source program in a directory different from the local one (''current working directory''), there is a question as to where ucode and icode files should be created: in the local directory or in the directory that contains the source program. On most systems, the appropriate place is in the local directory (the user may not have write permission in the directory that contains the source program). However, on some systems, the directory that contains the source file is appropriate. By default, the directory for creating new files is the local directory. The other choice can be selected by adding

```
#define TargetDir SourceDir
```

(SourceDir is defined in config.h before define.h is included there.)

Command-line options: The command-line options that are supported by icont are defined by Options. The default value (see config.h) will do for most systems, but an alternative can be included in define.h.

Similarly, the error message produced by icont for erroneous command lines is defined by Usage. The default value, which should correspond to the value of Options, is in config.h, but may be over-ridden by a definition in define.h.

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

Character set: If you are porting Icon to a computer that uses the EBCDIC character set, add

```
#define EBCDIC
```

to define.h.

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 define.h.

Host identification: The identification of the host computer as given by the Icon keyword &host needs to be specified in define.h. The definition

```
#define HostStr "unspecified host"
```

is provided in define.h initially. This definition should be changed to an appropriate value for your system.

Miscellaneous: There are two other definitions that may be needed in some cases:

Hz	(default: 60)
UpStack	(default: undefined)

If you are running in a 50-hz environment, add

```
#define Hz 50
```

to define.h.

Most computers have downward-growing C stacks, for which stack addresses decrease as values are pushed. If you have an upward-growing stack, for which stack addresses increase as values are pushed, add

```
#define UpStack
```

to define.h.

Keyboard functions: If your system supports the keyboard functions *getch()*, getche(), and *kbhit()*, add

```
#define KeyboardFncs
```

to define.h to enable them.

Optional features: The implementation of co-expressions and arithmetic overflow checking require assembly language routines. Initially, define.h contains

```
#define NoCoexpr
#define NoOver
```

These definitions disable co-expressions and arithmetic overflow checks. Leave these definitions in for the first round, although you may want to remove them later and implement these features (see Section 7).

5.2 Operating System Differences

Conditional compilation for operating systems usually is due to differences in run-time library routines, differences in file naming, the handling of input and output, and environmental factors.

The presently supported operating system are AmigaDos, Atari ST TOS, the Macintosh under MPW, MS-DOS, OS/2, UNIX, and VM/CMS, and VMS. There hooks for transporting to an unspecified system (a new port). The associated defined symbols are

AMIGA	AmigaDos
ATARI_ST	Atari ST TOS
HIGHC_386	MS-DOS in 32-bit protected mode for 80386 processors
MACINTOSH	Macintosh
MSDOS	MS-DOS
MVS	MVS
OS	OS/2
PORT	new port
UNIX	UNIX
VM	VM/CMS
VMS	VMS

Conditional compilation uses logical expressions composed from these symbols. An example is:

```
                .
                .
                .
#if MSDOS
                .
                .               /* code for MS-DOS */
                .
#endif

#if UNIX || VMS
                .
                .               /* code for UNIX and VMS */
                .
#endif
                .
                .
                .
```

Each symbol must be defined to be either 1 (for the target operating system) or 0 (for all other operating systems). This is accomplished by defining the symbol for the target operating system to be 1 in define.h. In config.h, which includes define.h, all other operating-system symbols are automatically defined to be 0.

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. Note that it is important not to use #ifdef accidentally in place of #if, since all the names are defined.

The file define.h initially contains

```
#define PORT 1
```

Leave it as is; later you should come back and change PORT to some more appropriate name.

Note: The PORT sections contain deliberate syntax errors (so marked) to prevent sections from being overlooked during porting. These syntax errors must, of course, be removed before compilation.

To make it easy to locate all the places where there is code that may be dependent on the operating system, such code is bracketed by unique comments of the following form:

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

Between these beginning and ending comments, the code for different operating systems is provided using conditional expressions such as those indicated above.

There presently are a total of 61 segments that contain such code. The files that contain operating-system-dependent code are listed in Appendix B. Look through some of the files that contain such segments to get an idea of what is involved. Each segment contains comments that describe the purpose of the code. In some 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 to take advantage of existing code for other systems, as in

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

#if VMS
        :
#endif
```

and removing the present code for PORT or by filling in the segment with the appropriate code, as in

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

If no code for the target operating system, a comment should be added so that it is clear that the situation has been considered.

You may find need for code that is operating-system dependent at a place where no such dependency presently exists. If this happens, add a new segment similar to the other ones, being sure to provide something appropriate for all operating systems. 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 in these segments; this encourages errors and obscures the mutually exclusive nature of operating system differences.

6. Building and Testing

6.1 The Command Processor

Start by compiling all the C programs listed in icont.bat. Link the resulting object files to produce icont. If you encounter problems, first check the portions of code containing operating system dependencies.

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

```
icont -c bitops.icn
```

Be careful to run icont in a way that does not overwrite the distributed ucode files in suite1.

The –c option stops icont at the point it produces *ucode* files, which are an intermediate form of virtual machine code. 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 unless the EBCDIC character set is used in the port.

Checking icode files is next. Since icode files are binary and vary somewhat from system to system, they cannot be checked as easily as ucode files. However, there is an option that produces diagnostic output in printable form. To obtain this diagnostic output, which has the suffix .ux, use icont with the –L option and a .u1 file, as in

```
icont -L bitops.u1
```

Compare the result to the distributed .ux file. Remember that differences are to be expected and the check is only a rough one.

If trouble is encountered in icont, additional debugging output can be obtained by adding

```
#define DeBugIcont
```

to define.h and recompiling icont. Note that define.h initially contains

```
#define Debug
```

This enables both the -L output and DeBugIcont. It can be removed after icont is known to be running properly. Some space is saved as a result.

6.2 The Executor

If you get this far without apparent problems, you are ready for the next 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:

```
icont hello.icn
iconx hello
```

If all is well, the last step should print out "hello world" and some identifying information. If it doesn't, the problem may be in either icont or iconx.

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 UNIX implementation using the following script:

```
for i in `cat $1.lst`
do
   rm -f local/$i.out
   echo Running $i
   icont -s $i.icn
   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
```

Something similar can be concocted for most 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:

check.lst	programs that may produce different results on different systems
coexpr.lst*	programs that use co-expressions
expr.lst	programs that contain a wide variety of expressions
gc.lst	programs that test garbage collection
icon.lst*	short but varied programs
other.lst*	programs that test additional features
over.lst	programs that test arithmetic overflow checking
version7.lst*	programs that test new features in Version 7.5

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.

Start with icon.lst. The output should be identical to that in the distributed .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 other.lst test some features that are not tested elsewhere. They should be treated like the programs listed in icon.lst.

Since storage management is one of the parts of Icon that is likely to give trouble, there are special storage-management tests in gc.lst. These programs run for a long period of time. One program may show a difference in output if the fixed-regions version of memory management is used, since it may run out of space.

The programs in coexpr.lst and over.lst use features that require assembly-language code. Save them for later.

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 [4] and how it is implemented [1]. A good debugger can be very helpful.

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

Note: 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. If you do not implement co-expressions, the only effect will be that Icon programs that attempt to use a co-expression will terminate with an error message.

All aspects of co-expression creation and activation are written in C in Version 7.5 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.

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 (C *longs*) 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 size of the array for saving the C state is 15. This number may be changed by adding

```
#define CStateSize n
```

to define.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)
long *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 */
                    :
      }
}
```

After you implement coswitch, remove the #define NoCoexpr from define.h.

To test your context switch, run the programs in coexpr.lst. Ideally, there should be no differences in the comparison of outputs.

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.

7.2 Overflow Checking

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.

Initially, define.h contains the definition

```
#define NoOver
```

which causes overflow checking to be bypassed.

If you do not want to implement overflow checking, you need do nothing. The only effect will be that overflow will not be detected.

If you want to implement overflow checking, remove the definition of NoOver from your define.h and write routines ckadd, cksub, and ckmul that call fatalerr(–203,0) in the case of overflow.

Note: It often is harder to test for overflow for multiplication than for addition and subtraction. A dummy routine that simply returns can be provided for multiplication if this is the case on your system.

To test overflow checking, run the programs in over.lst. There should be no differences in the comparison of outputs if overflow checking is working properly. 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 Building
The University of Arizona
Tucson, AZ 85721
U.S.A.

(602) 621-2018

icon-project@arizona.edu (Internet)
... {uunet, allegra, cmcl2, 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. 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.

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 7.5 of Icon on UNIX Systems*, The Univ. of Arizona Tech. Rep. 88-6a, 1988.
3. R. E. Griswold, G. M. Townsend and K. Walker, *Version 7 of Icon*, The Univ. of Arizona Tech. Rep. 88-5, 1988.
4. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
5. R. E. Griswold, *An Overview of the Icon Programming Language*, The Univ. of Arizona Tech. Rep. 83-3h, 1983.
6. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
7. Technical Committee X3J11, *Draft Proposed American National Standard for Information Systems — Programming Language C*, 1988.

Appendix A — Files Used for Components of Icon

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

Files Used for icont

config.h*	general configuration information
cpuconf.h*	processor configuration information
define.h*	system-dependent definitions
fdefs.h*	function definitions
general.h	general header information
globals.h	global declarations
header.h*	icode header structure
keyword.h*	keyword definitions
lfile.h	information for link declarations
link.h	heading information for the linker
odefs.h*	operator definitions
opcode.h	opcode structure
opdefs.h*	icode instruction definitions
paths.h*	file paths
rt.h*	header for run-time system
sizes.h	data sizing
tlex.h	information for lexical analysis
token.h	token definitions
trans.h	heading information for the translator
tree.h	code tree information
tsym.h	information for symbol tables
version.h*	version information
common.c*	routines common to icont and iconx
err.c	error messages
keyword.c	keyword structure
lcode.c	code generator
lglob.c	processor for global linking information
link.c	linker
llex.c	lexical analyzer
lmem.c	linker memory management
lnklist.c	file linking
lsym.c	linker symbol table management
opcode.c	opcode table
optab.c	state tables for operator recognition
parse.c	parser
tlex.c	lexical analyzer for translation
tlocal.c	local routines
tmain.c	main program
tmem.c	memory management for translation
toktab.c	token table
trans.c	translator
tree.c	code tree constructor
tsym.c	translator symbol table management
util.c	utility routines

Files Used for iconx

config.h*	general configuration information
cpuconf.h*	computer configuration information
define.h*	system-dependent definitions
fdefs.h*	function definitions
gc.h	garbage collection definitions
header.h*	icode header
keyword.h*	keyword definitions
memsize.h*	memory sizing
odefs.h*	operator definitions
opdefs.h*	icode definitions
rt.h*	run-time definitions
version.h*	version information
common.c*	routines common to icont and iconx
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
rdebug.c	debugging routines
rdefault.c	default value routines
rdoasgn.c	assignment routines
rlocal.c	local routines
rmemexp.c	memory management routines for expandable regions
rmemfix.c	memory management routines for fixed regions
rmemmgt.c	general memory management routines
rmisc.c	miscellaneous routines
rstruct.c	structure routines
rsys.c	system routines

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.

h:

- config.h (1)
- rt.h (1)

icont:

- lcode.c (1)
- link.c (4)
- lmem.c (4)
- tlocal.c (1)
- tmain.c (4)
- trans.c (2)
- util.c (1)

iconx:

- fsys.c (12)
- idata.c (1)
- imain.c (10)
- interp.c (4)
- lmisc.c (6)
- rconv.c (1)
- rlocal.c (1)
- rmemexp.c (2)
- rmemmgt.c (1)
- rmisc.c (1)
- rsys.c (3)