Transporting the Icon Programming Language*
Version 2.0

Ralph E. Griswold, David R. Hanson,
and Stephen B. Wampler

TR 79-2b

March 1979
Revised June 1979 and February 1980

Department of Computer Science
The University of Arizona

*This work was supported by the National Science Foundation under Grants MCS78-02545 and MCS79-03890.

Transporting the Icon Programming Language

1. Overview

Most of the Icon programming language is implemented in Ratfor [1], a preprocessor that generates Fortran code. As a result, implementing Icon on a new computer is largely a matter of compiling all the Ratfor code locally.

There are two components of the Icon system — a translator and a runtime system. The translator converts Icon procedures into Fortran subroutines. These subroutines contain calls to runtime routines that are executed to carry out the operations specified in the Icon program. The subroutines that are produced by the translator are then compiled by a standard Fortran compiler and the result is linked with the runtime system and executed.

There are a few machine-dependent parameters in Ratfor include files that must be set for your computer. You must also write a few machine-dependent routines that cannot be written in a machine-independent way in Ratfor. Depending on your local character set, you may have to build a modified version of the Ratfor input/output system to accomodate Icon's internal character set. In addition, you may have to modify the main programs for the Icon translator and runtime system.

The remainder of this report describes the process of transporting Icon in more detail.

2. Distribution Material

The distribution material for the portable Icon system consists of a magnetic tape and a number of documents.

The recording format of the distribution tape is described on a label on the tape reel. The tape contains the following files:

1. rdearc: Ratfor routine for separating archive files
2. fdearc: Fortran routine for separating archive files
3. includ: include files for the Icon translator and runtime system
4. trans: Ratfor routines for the Icon translator
5. runt: Ratfor routines for the Icon runtime system
6. mdepen: test programs for the machine-dependent routines
7. itest: Icon test programs
8. idata: data for the Icon test programs
9. ftest: generated Fortran for the Icon test programs
10. iresul: results of running the Icon test programs
11. rtest: Icon programs in limited character set
12. ebcdic: block data Ratfor routine for EBCDIC character set mapping
13. xref: reference information

The names of the files are used for reference in this document and are not on the tape itself. All files except rdearc and fdearc are in an archive format [1], with headers separating sections (the alternative would have been to distribute a tape containing hundreds of files). You therefore will have to separate the files as appropriate for your local installation. To help in separating the archive files, rdearc and fdearc contain prototype routines that can serve as a basis for a program to do the separation. rdearc is written in Ratfor and fdearc is the equivalent Fortran routine. These routines are only prototypes — you will have to modify them to suit you local system. The header names in the archive files correspond to file names used at our installation. In some cases, such as for include files, the header names correspond to files explicitly referenced in routines. In other cases, they serve only as mnemonic guides.

There are five routines (ESCAPE, FTOC, ITOC, LENGTH, and PUTLIN) that are common to the translator and the runtime system. These are included in both trans and runt for convenience, in case you wish to keep the two components of the system separate.

The reference material in xref supplements the material given in the appendices of this document. You may find some of this material useful in building tools to help in the installation process.

The documents included with the distribution are:

1. directory of the distribution tape
2. transporting instructions (TR 79-2b, this document)
3. machine-dependent components of the DEC-10 and CDC implementations
4. Icon implementation notes (TR 79-12a)
5. description of the Icon storage management system (TR 78-16a)
6. installation instructions for the DEC-10 implementation of Icon
7. installation instructions for the CDC implementation of Icon
8. Icon reference manual (TR 79-1a)
9. summary of the differences between Versions 1 and 2 of Icon
10. user's guide for the DEC-10 implementation of Icon
11. user's guide for the CDC implementation of Icon

The reference manual is copyrighted. Permission to reproduce this manual for your local use is granted in the covering letter for the distribution material. The other documents are not copyrighted and you may reproduce them freely.

The user's guides for the DEC-10 and CDC implementations of Icon are included as models for documentation that you may wish to provide for the users of your implementation. The installation instructions for the DEC-10 and CDC implementations are included because they contain examples of how other implementations are organized and how machine-dependent problems have been handled elsewhere.

3. Potential Problems

Before undertaking the implementation of Icon on your computer, you should determine the feasibility of the implementation. Problems are most likely to arise in the following areas:

1. capacity limitations
2. software inadequacies
3. hardware properties
4. operational and organizational limitations

3.1 Capacity Problems

The one factor that is most likely to make the implementation of Icon impractical or even unfeasible is limited memory capacity. Icon is a large system and cannot be easily adapted to run in a small amount of memory. Memory requirements can be estimated from the amount of memory used in existing implementations; some figures are given in Appendix A. In interpreting these figures keep two points in mind: (1) Icon data layouts are based on Fortran integers, and (2) these are implementations for which no attempt was made to minimize memory usage. Note also that the amount of memory required to *construct* components of Icon may be larger than the amount of memory needed to *run* them.

Since various internal tables, data layouts, and source-language objects are composed of Fortran integers, the amount of space for data scales down for machines with small word sizes. The amount of space required for executable code may not scale down correspondingly.

In situations where the amount of available memory is marginal, various internal tables and buffers can be reduced in size. Such reductions may restrict the class of Icon programs that can be run and may also reduce running speed because of increased overhead for storage management. A list of places where reductions may be made is given in Appendix B.

A more extreme approach is to subset the Icon system by removing language features. Since the Icon system is highly modularized, this can be done by eliminating parts of the runtime system on a per-routine basis. Appendix C contains a list of features that can be removed without substantially diminishing the usefulness of the language. If Icon is subsetted in this fashion, the translator will still accept the deleted features, but their use will cause unresolved references when the resulting Fortran program is linked with the runtime system. This problem should be tolerated; we do not advise modifying the translator.

3.2 Software Inadequacies

In the software area, a decent Fortran IV compiler is essential.

There are four areas in Icon translator and runtime system in which Fortran may cause problems:

1. inability to compile large routines
2. inability to handle a large number of common blocks
3. inability to handle Fortran constructions used in the coding of Icon
4. unavailability of intrinsic Fortran routines

There are two very large routines in the translator: CODGEN and PARSE. If these routines exceed the capacity of your Fortran compiler, there is no easy solution.

Icon uses several named common blocks, 8 in the translator and 13 in the runtime system. See Appendix D. Blank common is used for the main memory array in the include file CMEM. If your Fortran compiler cannot handle these common blocks, you can combine or revise them (we know of no name collisions, but this is a possible problem). Combining the common blocks is a substantial undertaking, since almost all the Ratfor routines will require editing.

Several Fortran constructions used in the coding of Icon are beyond the official ANSI standard. However, they fall within the common *de facto* implementation standard. These include:

1. use of arbitrary expressions (but not function calls) in array subscripts and computed gotos
2. passing arguments that are variables in a labeled common of the subroutine being called
3. passing the same variable as two different arguments

Problems with Fortran in an Icon translator or runtime routine usually can be solved my making appropriate changes to the Ratfor code or to the corresponding Fortran code for the routine. Problems with Fortran in the code generated by the Icon translator are likely to be more difficult to fix, since it may be necessary to change the logic of the translator. With two exceptions, the Fortran code generated by the Icon translator conforms to the Fortran standard as embodied in the PFORT verifier [4].

The most serious exception occurs in the initialization of arrays. It is assumed that arrays can be initialized by data statements of the form

```
data a /value of a(1), value of a(2), ..., value of a(N)/
```

or, if the array is large, by data statements of the form

```
data (a(i),i=1,100) /value of a(1), ..., value of a(100)/
data (a(i),i=101,200) /value of a(101), ..., value of a(200)/
data (a(i),i=201,N) /value of a(201), ..., value of a(N)/
```

The ANSI standard form requires explicit specification of each element of the array, i.e.

```
data a(1) /value of a(1)/
data a(2) /value of a(2)/
          .
          .
          .
data a(N) /value of a(N)/
```

If the form of the data statements causes problems, it can be changed by modifying the translator routine outds, which is called to output most data statements. Exceptions are the array r, which is output in outhdr, and field offset arrays, which are described below.

The second exception concerns the use of block data subprograms. If records are used in an Icon program, arrays containing field offsets are generated and placed the labeled common cflds. A block data subprogram is generated that initializes these arrays. The problem is that, in this case, there are two block data subprograms: the one that initializes cflds and one that initializes other runtime data used by every Icon program. Having more than one block data subprogram is contrary to the ANSI standard and may cause problems. If so, the offset arrays can be made local to each Fortran procedure (corresponding to each Icon procedure) by modifying the translator routine outfld. This routine is called to output the common statement in each subroutine and may be modified to output the data statements in place of the common statement. Note that the arrays are output directly by outfld; outds is not called. Thus, if the form of data statement mentioned above causes problems, outfld will need to be modified.

In addition, the translator does not provide range checks on the integer variables of the computed gotos that it generates. In theory, such values should always be in range. However, if an out-of-range value should occur, the source of the error may be difficult to locate, since such a branch is not defined in the ANSI standard.

If the translator generates more statement continuations in DATA statements than your Fortran compiler allows, reduce the define constant CUTOFF in the translator include file TDEF. (This constant specifies the number of elements, not the number of lines.)

If your Fortran compiler does not support all of the intrinsic routines used by Icon (see Appendix E), this will probably show up as a linking error. All the intrinsic functions used by Icon are simple, and you easily can provide local assembly language routines for them. EXTERNAL declarations may be necessary.

Your linker loader must be able to handle a large number of routines and must be able to resolve a large number of references. It is useful, but not necessary, for the linker/loader to be capable of searching a library. This capability allows loading only those runtime routines needed by a particular Icon program, thus generally reducing runtime memory requirements. A complete list of Ratfor routines is given in Appendix F.

3.3 Hardware and Architectural Problems

If your computer has a small word size, you may have some problems. For the Icon programmer, integer arithmetic may be uncomfortably limited in range.

The Icon system presently assumes that Fortran real numbers occupy the same amount of space, and have the same alignment, as Fortran integers. If these assumptions are not valid, various malfunctions may occur when real arithmetic is performed in an Icon program. There is no easy solution to this problem at the moment, although we are working on one for future versions.

3.4 Operational Problems

The overall size of the Icon system, its complexity, and extensive modularization may present significant problems in some environments. Unless you have a large amount of disk space and a good file system, the implementation of Icon may be very difficult in practice, even if all other resources are adequate. It is worth spending some time in planning the procedures that you will use for assembling, modifying, and maintaining the system.

4. The Transporting Process

We suggest implementing Icon in the following steps:

1. implement and test Ratfor
2. make a version of the Ratfor input/output library for Icon
3. set machine-dependent parameters
4. implement and test the machine-dependent Icon routines
5. build and test the translator
6. build the Icon runtime system
7. build the runtime library
8. test the entire Icon system

4.1 Implementing Ratfor

The implementation of Ratfor is essentially a separate process. The version of Ratfor that is used for the implementation of Icon must support the return statement and the string declaration. The size of its define table, given by the constant MAXTBL, must be at least 6100. If you do not already have a suitable version of Ratfor, you may obtain one from

David R. Hanson
Department of Computer Science
University Computer Center
Tucson, Arizona 85721 USA
Telephone: (602)-626-3617

In addition to the Ratfor preprocessor, the Ratfor input/output system is used extensively by Icon and must work properly. We recommend that the Fortran version of the input/output system be used initially, although the improvement in performance obtainable through use of machine-dependent input/output routines may dictate a change to machine-dependent routines at a fairly early stage.

4.2 Revising the Ratfor Input/Output System

The Ratfor input/output system needed to support Icon must accomodate Icon's internal character set. The size of Icon's internal character set is 256, of which the first 128 characters correspond to ASCII [2]. Appendix G contains a list of ASCII codes and graphics. The remaining 128 characters are available to the Icon programmer and may be used for a variety of purposes. The size and interpretation of the internal character set is independent of the external character set of your computer. Since the Ratfor input/output system is based on ASCII, the only problem that may be encountered is in dealing with the extra 128 characters in Icon's internal character set. It is generally possible to configure the Ratfor input/output system to work for Icon as well as for Ratfor programs. The following paragraphs describe character mappings in more detail.

The translation between your external character set and the internal one is done in the Ratfor input/output system. In the basic Ratfor system, translation between your character set and the internal character set is done by a table lookup in INTCHR and EXTCHR. This lookup is based on the interpretation of Fortran literals on your computer and supports a natural interpretation of the 96-character graphic subset of ASCII. As described in the Ratfor installation instructions, you should replace these functions by arrays that index codes according to your character set. Select a mapping that is appropriate to Icon's concept of its internal character set and its relation to your computer. Note that Ratfor stores characters unpacked as Fortran integers, so the size of Icon's internal character set is not a problem.

For computers for which ASCII is the natural character set, the translation is trivial. On input, map characters directly into their internal code. On output, map characters in the first half of the internal character set directly, but fold characters in second half to corresponding positions in the first half by ignoring the high-order bit. (This treatment of the upper half of the character set is not essential, but it is reasonably natural and is used in existing ASCII implementations of Icon.)

For character sets that are smaller than ASCII, use a natural mapping on input. Where there are common graphics, map the external characters into the ASCII codes for those graphics (note that this may produce an internal collating sequence that is different from the normal collating sequence on your computer). See the CDC installation guide for an example.

Since Icon relies on ASCII codes, failure to preserve graphic correspondences may cause the malfunction of Icon programs. Some source-language features, such as &lcase and &ucase, are particularly sensitive to this problem. On output, perform the inverse mapping for the internal characters that correspond to external ones. For internal characters that do not have external correspondences, provide reasonable mappings. For example, if your character set is BCD, map internal lower-case characters into upper-case characters on output.

For the EBCDIC character set [3], a recommended mapping is given in a Ratfor block data routine ebcdic on the distribution tape. This mapping is 1-to-1, so that external data can be processed without loss of information. In addition, ASCII codes have been selected to correspond, where possible, to the graphics on the IBM extended TN print train [3]. Where no graphic correspondence or equivalent interpretation exists, codes have been selected to preserve the inverse property.

For character sets other than those listed above, the same general guidelines follow. For example, if Icon is to be implemented on a computer with a character set size larger than 256, the input must be folded.

4.3 Machine-Dependent Parameters

There are a number of machine-dependent parameters that you need to set according to the architectural characteristics of your computer. These parameters appear near the beginning of the include files TDEF and IDEF and are clearly marked. The values of the parameters in the files as distributed are those for the DEC-10. For reference, listings of the DEC-10 and CDC parameters are included in the distribution documentation.

Examine these parameters carefully and change them to values appropriate to your computer. *Failure to set these parameters correctly may produce catastrophic results.*

4.4 Machine-Dependent Routines

The eight machine-dependent routines needed to supplement the Ratfor component of Icon are described in Appendix H. For reference, listings of the DEC-10 and CDC implementations of these routines are included in the distribution documentation. The first five routines (LLC, LDC, STC, SETB, and TSTB) manipulate characters and bits. The proper functioning of these routines is essential. A stand-alone test program for these routines is included in mdepen on the distribution tape. Note that you must set machine-dependent parameters in this test program. The other three routines (SYSERR, RUNTIM, and DATE4) interface the operating system. They may be implemented by dummy routines as an initial stop-gap measure.

4.5 Icon Translator

The Ratfor component of the Icon translator is trans on the distribution tape. Appendix F contains a directory of these routines.

You may need to modify the translator's main routine, TMAIN, to redirect program listings (see TRNLAT). You may also need to open files or connect standard input and output according to the conventions of your Fortran compiler and operating system. To change the heading produced for program listings, modify PUTLIST. If you encounter problems in processing the translator Ratfor routines through your version of Ratfor, there is probably a problem with Ratfor itself. Trouble encountered in compiling the resulting Fortran code may come from several sources. See Section 3.2 for a list of the most likely problems.

Once the Ratfor routines for the translator are successfully compiled, link them with the Ratfor input/output system and the machine-dependent routines. Problems during linking may again come from a variety of sources. Unresolved references may indicate use of intrinsic Fortran routines that are not available locally. If you have an extended Ratfor input/output system, you may have routines with the same names as those in the translator or in the machine-dependent routines. Delete or override any such routines in your Ratfor input/output system.

Run the translator on the Icon test programs contained in itest on the distribution tape. If you have a limited character set or restricted input/output devices, you may wish to use use rtest, which uses an approximation to the PL/I character set, rather than itest, which uses full ASCII.

The output of the translator, a subroutine named ICON, should be syntactically correct (but highly stylized) Fortran code. If the translator malfunctions, the most likely source of error is in the machine-dependent routines that you supplied locally. Compare the Fortran code generated by your translator with the Fortran code in ftest on the distribution tape. There should not be any differences unless you change the text of the test programs (see Section 4.8).

4.6 Runtime System

The Ratfor component of the runtime system is given in runt on the distribution tape. A directory of these routines is contained in Appendix F.

You may need to modify the runtime system main routine, RMAIN, in the same manner as the translator main routine.

Compile the Ratfor routines for the runtime system. The same kinds of problems may arise here that may arise in the translator.

4.7 Runtime Library

Combine the runtime routines with the Ratfor input/output system and the machine-dependent routines to form a library. Be sure to delete or override any routines in your Ratfor input/output system whose names are the same as those in the runtime system or machine-dependent routines. Link this library with the result of translating an Icon program. As a first step, the entire runtime system should be included, although reduced memory utilization may be obtained by searching the library of runtime routines to link only those required by a particular program (see Section 5). For some systems, the library may have to be topologically sorted. A list of the runtime routines in topological order is contained in xref on the distribution tape.

4.8 Testing the Entire Icon System

Test the entire Icon system beginning with the first test program. If all goes well, test the more complicated programs. The first five test programs are self contained and require no data. The remaining programs require data contained in idata on the distribution tape. Appendix I contains a list of the test programs and data. When comparing the results of running the test programs with the results given in iresul, allow for differences that may result from different character sets or different listing formats. For example, the program wordt behaves differently when processing all upper-case data than when processing upper- and lower-case data. Thus itest and rtest give dirrerent results in some cases. Note that the test programs do not exercise all the features of Icon. A more comprehensive set of tests in being developed.

There are several possible sources of error that may be in the Icon system itself rather than in your implementation. Unless an error is obviously of such a nature, local implementation problems should be suspected, since Icon has been in use for over two years at the University of Arizona. However, "intrinsic" errors are certainly possible. For example, we may have overlooked a machine dependency. There are also certainly some errors in the logic of the Icon system. Such errors, however, probably will not appear in running the test programs, since these programs are known to run satisfactorily on the two implementations mentioned above.

5. Improving the Implementation

The most dramatic improvement in the performance of Icon can be obtained by replacing Fortran components of the Ratfor input/output system by locally tailored machine-dependent routines. In fact, you probably will have to do some work in this area to obtain tolerable running speeds. Information on this is contained in our Ratfor distribution material.

If your linker/loader has a good library search capability, you may be able to reduce the memory requirements for running Icon programs by linking only those runtime routines that are actually referenced by the Fortran program produced by the translator. Caution is advised here: the time required to search the runtime library may be unacceptably large.

A more drastic approach to improving the performance of Icon is to replace Ratfor routines by corresponding assembly-language routines. The best candidates for this kind of improvement are those that are fairly simple, but used frequently. Performance measurements of Icon indicate that the prime candidates for replacement are the routines MVC and MVW.

Warning: do not attempt to improve the Fortran code by use of LOGICAL *1; the results will be catastrophic.

6. Extending the Language

The way Icon is implemented makes extensions difficult. However, we have made a provision for the inclusion of locally supplied functions that may be useful for such purposes as interfacing your operating system.

The translator recognizes ZZ0, ZZ1, ... ZZ9 as the names of built-in functions. Runtime routines for these functions are not included in the Icon system; however they may be provided locally. Appendix J contains an example of such a routine.

Any extension of a nontrivial nature requires considerable knowledge of the internal workings of Icon. Documentation of the implementation of Icon is included in the distribution package.

7. Inaccessible Code

There is some performance measurement code in the Icon system that is inaccessible unless modifications are make to allow users to set switches when Icon is run. This code may also contain machine dependencies. Although this code adds somewhat to the size of the Icon system, it was not removed from the portable system because to have done so would have been a substantial undertaking and might have introduced errors.

8. Feedback, Maintenance, and Updates

It will be very helpful, especially to other implementors, to have your comments, criticisms, and reports of problems at the earliest possible time.

As described in the covering letter for the distribution package, you may access our computer system to communicate with us directly through our message facility. If you do not use our computer system, send information to

Ralph E. Griswold
Department of Computer Science
University Computer Center
The University of Arizona
Tucson, Arizona 85721 USA
Telephone: (602)-626-1829

As troubles are discovered, we will distribute information and additional documentation automatically. If you access our computer system, this information will be available when you log in. Otherwise, we will mail it to you.

We will distribute minor updates to program material in whatever form seems most appropriate. We will distribute major updates in the form of a new distribution tape. For this reason, you should keep careful records of any changes you make, especially to any Ratfor routines in the machine-independent portion of the system. We recommend that you use some uniquely identifiable commenting convention for changes that you make locally.

Acknowledgement

Tim Korb played a major role in the design and execution of the implementation of Icon. Cary Coutant has assisted with the preparation of the system for transporting. The experience of installers of earlier versions of this system has resulted in a number of improvements contained in the current system. Special thanks are due to William H. Mitchell, Christopher St. James, and Michael D. Shapiro.

References

1. Kernighan, Brian W. and P. J. Plauger. *Software Tools.* Addison-Wesley, Reading, Massachusetts. 1976.

2. American National Standards Institute. *USA Standard Code for Information Exchange*, X3.4-1977. New York, New York. 1977.

3. IBM Corporation. *System/370 Reference Summary*. Form GX20-1850-3. White Plains, New York. 1976.

4. Ryder, B. G. *Software — Practice and Experience*, Vol. 4, No. 4 (December 1974), pp. 359-377.

Appendix A—Space Requirements

The space used by two implementations of Icon for two test programs is shown below. The program hello is trivial; the program rsg is moderately large and uses many of the features of Icon. The figures are approximate.

	hello		rsg	
	kilowords	megabits	kilowords	megabits
DEC-10				
translation	33.3	1.2	33.3	1.2
compilation	26.2	1.0	28.2	1.1
linking	37.9	1.4	39.9	1.5
execution[1]	33.3	1.2	36.9	1.3
execution[2]	23.6	0.8	31.8	1.1
CDC Cyber/6000				
translation	23.9	1.4	23.9	1.4
compilation	21.5	1.3	21.5	1.3
linking	16.4	1.0	25.6	1.5
execution[2]	17.3	1.0	23.1	1.4

[1]linking entire runtime library
[2]searching runtime library

Appendix B—Parameter Settings

The following parameters may be lowered to reduce memory requirements or raised to increase the capacity of the Icon system. The minimum values given are our recommendations. Lower values may be used, but the result may be unsatisfactory.

Translator (TDEF)

parameter	current	minimum	affected property
MAXHEAP	1500	1000	translator work area
MAXSTACK	300	200	translator stack size
MAXGBLS	100	50	number of global symbols
MAXREALS	100	10	number of real numbers
MAXINTGS	100	50	number of integer literals
MAXSTRGS	500	100	number of string literals
MAXPROCS	1000	80	size of procedure area
MAXSYMTB	2000	1000	symbol table size
MAXRECDS	200	50	number of record types
MAXLEVEL	8	0	number of levels of includes
MAXLABELS	1024	256	number of local labels in a procedure

Runtime System (IDEF)

MEMSIZE	15000	4000	dynamic memory space
STRSIZE	1000	500	string storage region size[1]
SQLSIZE	200	10	string qualifier storage region size[1]
INTSIZE	200	10	integer storage region size[1]
HEPSIZE	4000	500	heap storage region size[1]
STKSIZE	200	50	stack size[1]
CSTACKSIZE	500	300	control stack size[1]

[1]The sum of these values must not exceed MEMSIZE. These are initial values; they are automatically adjusted during execution and the regions expand as needed.

Appendix C—Subsetting Suggestions

Some features of Icon may be removed because they can be easily written in Icon itself or because they are not essential for most programs. Subsetting is somewhat a matter of taste. Our suggestions follow. In some cases, such as for stack functions, features should be removed or retained as a group.

Features Easily Written in Icon

	Routine to Remove
center(s1,i,s2)	XCENT
left(s1,i,s2)	XLEFT
mod(i,j)	XMOD
numeric(x)	XNUMR
pop(k)	XPOPS
pos(i,s)	XPUSHS
push(k,x)	XPUSHS
random(i)	XRAND
repl(s,i)	XREPL
reverse(s)	XREV
section(s,i,j)	XSECT
top(k)	XTOPS
=s	XTABM
c1 -- c2	XDIFF
c1 ** c2	XINTER

Inessential Features

copy(x)	XCOPY
display(i)	XDISP
trim(s,c)	XTRIM
x :=: y	XSWAP
x <->	XRSWAP
i ^ j	XPOWER

Appendix D—include files

The include files are separated into definition files and common blocks.

Definition Files

ADEF	ASCII character mnemonics
CDEF	definitions common to PARSE and CODGEN
IDEF	runtime system definitions
KDEF	keyword token definitions
LDEF	lexical definitions
ODEF	token definitions for OPCODE
PDEF	label definitions for PARSE
SDEF	symbol table field defintions
TDEF	translator definitions

Common Blocks

CCODE	generated code information
CCSTK	control stack common
CFOLD	equivalence map for translator folding
CGC	garbage collector common
CINTP	interpreter common
CIO	input output common for runtime system
CLAB	global labels for translator
CLEX	global variables for lexical analyzer
CMAIN	interface to generated Fortran code
CMEM	Icon memory common
CPARM	runtime parameters
CSIZES	initial sizes for storage region
CSTAT	common for storage management statistics
CSYM	symbol table structure for translator
CTEND	variables tended by garbage collector
CTIO	input output common for translator
CTRANS	global variables for translator
CUTIL	utility common
CXMAP	character map table
CXSORT	field offset for sorting

Appendix E—Intrinsic Fortran Functions Used by Icon Ratfor Routines

1. ABS(*real*), returning the absolute value of *real*.

2. IABS(*integer*), returning the absolute value of *integer*.

3. FLOAT(*integer*), returning real equivalent of *integer*.

4. IFIX(*real*), returning the largest integer less than or equal to the absolute value of *real* and with the sign of *real*.

5. AMOD(*real1,real2*), returning the real remainder of dividing *real1* by *real2*.

6. MOD(*integer1,integer2*), returning the integer remainder of dividing *integer1* by *integer2*.

7. MAX0(*integer1,integer2*), returning the larger of *integer1* and *integer2*.

8. MIN0(*integer1,integer2*), returning the smaller of *integer1* and *integer2*.

Lists of routines in which Fortran intrinsic functions are used are included in xref on the distribution tape.

Appendix F—Directory of Ratfor Routines

Translator Ratfor Routines

ALCLAB	allocate label
ALCNOD	allocate parser node
BADERR	issue fatal error and terminate
BIFNC	identify built-in function
BIOPR	identify built-in operator
CODGEN	generate code from parse tree for procedure
CTOF	convert character to real
CTOI	convert character to integer
CVARG	add argument conversion nodes for built-in function
CVNMX	add numeric conversion nodes for exponentiation
CVNUMR	analyse mixed-mode arithmetic expression
CVSTRG	add string conversion nodes
DEFLT	default parse tree for argument of built-in function
DEFTYP	default argument for built-in function
DOGLOB	parse global declaration
DOINCL	increment include level and open include file
DORECD	parse record declaration
ENTDCL	enter explicit scope declarations
ENTDEF	enter undeclared identifier
ENTDYN	enter dynamic identifier
ENTER	enter identifier into symbol table
ENTGBL	enter global identifier
ENTLCL	enter local identifier
ENTPAR	enter procedure parameters
ENTPRO	enter procedure
ENTSV	enter static identifier
ESCAPE	interpret literal escape conventions
FINDP	find procedure
FNARG	type or default for argument of built-in function
FNFAIL	check failability of function or keyword
FNTYP	type of function or keyword
FORLAB	allocate Fortran label
FTOC	real to character conversion
GENER	distinguish generators
GETALF	get alphanumeric token
GETKEY	get keyword token
GETLIN	get line
GETNUM	get numeric token
GETOPR	get operator token
GETSTR	get string literal
GETTOK	get token
IGNORE	determine token to ignore in event code
INOPR	determine if token is possible infix operator
INSERT	insert string into symbol table
ITOC	integer to character conversion
KEYTYP	determine type of keyword
KEYWRD	identify keyword
LENGTH	compute length of string
LEX1	get token during pass 1
LEX2	get token during pass 2
LOKFLD	look up identifier in record definition table

LOKGBL	look up pointer in global table
LOKLCL	look up pointer in local table
LOKREC	look up name in record table
LOKSTR	look up string
NGETC	get character
NODE1	allocate parse node with 1 field
NODE3	allocate parse node with 3 fields
NODE4	allocate parse node with 4 fields
NODE5	allocate parse node with 5 fields
NODE6	allocate parse node with 6 fields
NODE7	allocate parse node with 7 fields
NODE8	allocate parse node with 8 fields
OPCHR	determine operator character
OPCODE	print subroutine call for operation
OPRVAL	determine value of operator during pass 1
OPTYP	determine type of operator during pass 1
OUTCH	output character
OUTDON	finish output line
OUTDS	output data statement for array
OUTFLD	output field offset array declaration and initialization
OUTHDR	output heading for generated Fortran program
OUTMSG	output message
OUTNUM	output number to generated Fortran code
OUTTAB	position Fortran code after column 6
OUTTOK	output token to scratch file
PARSE	create parse tree
PASS1	pass 1 of translation
PBSTR	push string back onto input
POP	pop value from translator stack
POP2	pop two values from translator stack
POP3	pop three values from translator stack
POP4	pop four values from translator stack
PRINTX	formatted output to generated Fortran code
PUSH	push value on translator stack
PUSH2	push two values on translator stack
PUSH3	push three values on translator stack
PUSH4	push four value on translator stack
PUTBAK	push character back onto input
PUTINT	output decimal number
PUTLIN	write string to file
PUTLST	start listing file
RCTOI	convert string to integer
RESERV	identify reserved word
RESGBL	resolve global references
SCNSYM	scan for identifiers
SEARCH	search table for name
SPANB	span stream of blanks
SYNERR	issue syntax error message
TDATA	translator data
TERMIN	determine terminating token
TINIT	initialize translator
TMAIN	translator main program
TRNLAT	translate Icon program

Directory of Runtime Ratfor Routines

ABUMP	increment allocation count
ADJUST	adjust pointer for sweep
ALCBLK	allocate block from heap
ALCINT	allocate integer
ALCSQL	allocate qualifier
ALCSTR	allocate space for string
BAL	find balanced string for bal(c1,c2,c3,s,i,j)
CMPFLD	compare fields of two structures for qsort
CMPSQL	compare qualifiers for qsort
COMPAR	compare source objects
CTOS	convert character buffer to Icon string
DHEAP	print dump of heap
DTTYPE	type of Icon object
DUPL	duplicate string
ERROR	error termination
ESCAPE	interpret literal escape conventions
EXCHAN	exchange values
EXPAND	expand storage region
FCLOSE	close file
FIND	find substring for find(s1,s2,i,j)
FOPEN	open file
FTOC	real to character conversion
GCHEAP	garbage collect heap
GCINT	garbage collect integer region
GCSQL	garbage collect qualifier region
GCSTK	garbage collect heap for stack room
GCSTR	garbage collect string region
HASH	compute hash number
IDATA	initialization data
IINIT	initialize storage
IMAGE	construct print image
INTMRK	mark accessible integers
ITOC	integer to character conversion
LENGTH	compute length of string
LXCMP	lexical comparison of strings
MARK	mark accessible blocks
MVC	move characters
MVW	move words
PIMAGE	print image
POSF	convert to positive position specification
PRINTF	formatted print
PUTLIN	write string to file
QSORT	sort array
RDPTR	redirect pointer for sweep
RMAIN	runtime main program
SAVE	save section of stack on control stack
SCHECK	check and dump storage statistics
SINIT	initialize symbol and literal tables
SIZE	size of block
SQLMRK	mark qualifier as accessible
STKCHK	check free space on stack
STOC	convert Icon string to character string
SWEEP	process all pointers in storage areas
TYPE	determine type of source object

TYPEV	determine type of object on stack
UNESC	escape sequence for character
UPTO	find character for upto(c,s,i,j)
XACC	structure access
XADD	i + j
XANY	any(c,s,i,j)
XASG	x := y
XBAL	bal(c1,c2,c3,s,i,j)
XBANG	!x
XCAT	s1 || s2
XCCSET	implicit conversion to cset
XCENT	center(s1,i,s2)
XCFILE	implicit conversion to file
XCINTG	implicit conversion to integer
XCLOSE	close(x)
XCMP	general comparison
XCNUMR	implicit conversion to numeric
XCOMP	compare literal
XCOPY	copy(x)
XCPROC	implicit conversion to procedure
XCREAL	implicit conversion to real
XCSET	cset(s)
XCSTAK	implicit conversion to stack
XCSTRG	implicit conversion to string
XDEREF	dereference argument
XDIFF	c1 -- c2
XDISP	display(i)
XDIV	i / j
XDRIVE	drive expression to success
XDUP	duplicate top item on stack
XECASE	check case expression failure
XEVERY	save stack data for every loop
XFACC	access field of record
XFIND	find(s1,s2,i,j)
XGLOBL	push global variable
XIMAGE	image(x)
XINTER	cs1 ** cs2
XINTG	integer(x)
XINVOK	invoke procedure
XKEYWD	return keyword
XLCMP	lexical comparison
XLEFT	left(s1,i,s2)
XLLIST	<x1,x2, ..., xn>
XLOCAL	push local variable
XLPBEG	enter loop
XLPEND	exit loop
XMANY	many(c,s,i,j)
XMAP	map(s1,s2,s3)
XMARK	mark stack heights and set failure
XMATCH	match(s1,s2,i,j)
XMLIST	list(i)
XMOD	mod(i,j)
XMOVE	move(i)
XMRECD	construct record
XMSTAK	stack(i)
XMTABL	table(i)
XMUL	i * j

XNCMP	numeric comparison
XNEG	-i
XNEXT	next iteration of loop
XNOTC	~c
XNULL	check null
XNUMR	numeric(x)
XOPEN	open(x,s)
XPACS	push &ascii onto stack
XPBCS	push cset(" ") onto stack
XPBLK	push blank onto stack
XPINTG	push integer onto stack
XPLPCS	push cset("(") onto stack
XPNCS	push cset(&null) onto stack
XPNULL	push &null onto stack
XPONE	push 1 onto stack
XPOP	pop argument off stack
XPOPS	pop(k)
XPOS	pos(i,s)
XPOWER	i ^ j
XPREAL	push real number onto stack
XPRPCS	push cset(")") onto stack
XPSTRG	push string onto stack
XPUSHS	push(k,x)
XPZERO	push 0 onto stack
XRAND	random(i)
XRASG	x <- y
XREAD	read(f)
XREAL	real(x)
XREPL	repl(s,i)
XRESET	reset procedure entry point
XRETRN	return from procedure
XREV	reverse(s)
XRIGHT	right(s1,i,s2)
XRSWAP	x <-> y
XSCN1	set up for string scanning
XSCN2	restore scanning environment
XSECT	section(s,i,j)
XSIZE	size(x)
XSORT	sort(x)
XSREAD	reads(f,i)
XSTOP	stop(s)
XSTRG	string(x)
XSUB	i - j
XSUBST	substr(s,i,j)
XSUSP	suspend procedure
XSWAP	x :=: y
XSWRIT	writes(f,s1, ..., sn)
XTAB	tab(i)
XTABM	=s
XTINVK	trace procedure invocation
XTO	i to j
XTOBY	i to j by k
XTOPS	top(k)
XTREF	table reference
XTRETN	trace return from procedure
XTRIM	trim(s,c)
XTYPE	type(x)
XUNION	c1 ++ c2
XUPTO	upto(c,s,i,j)
XWRITE	write(f,s1, ..., sn)

Appendix G—The ASCII Character Set

pos.	octal	graphic	ASCII keyboard seq.	function	pos.	octal	graphic	ASCII keyboard seq.	function
1	000		control shift P	null	65	100	@	@	
2	001		control A		66	101	A	shift A	
3	002		control B		67	102	B	shift B	
4	003		control C		68	103	C	shift C	
5	004		control D		69	104	D	shift D	
6	005		control E		70	105	E	shift E	
7	006		control F		71	106	F	shift F	
8	007		control G	bell	72	107	G	shift G	
9	010		control H		73	110	H	shift H	
10	011		control I	horizontal tab	74	111	I	shift I	
11	012		control J	line feed	75	112	J	shift J	
12	013		control K	vertical tab	76	113	K	shift K	
13	014		control L	form feed	77	114	L	shift L	
14	015		control M	carriage return	78	115	M	shift M	
15	016		control N		79	116	N	shift N	
16	017		control O		80	117	O	shift O	
17	020		control P		81	120	P	shift P	
18	021		control Q		82	121	Q	shift Q	
19	022		control R		83	122	R	shift R	
20	023		control S		84	123	S	shift S	
21	024		control T		85	124	T	shift T	
22	025		control U		86	125	U	shift U	
23	026		control V		87	126	V	shift V	
24	027		control W		88	127	W	shift W	
25	030		control X		89	130	X	shift X	
26	031		control Y		90	131	Y	shift Y	
27	032		control Z		91	132	Z	shift Z	
28	033		control shift K	escape	92	133	[	[	
29	034		control shift L		93	134	\	\	
30	035		control shift M		94	135	]	]	
31	036		control shift N		95	136			
32	037		control shift O		96	137	-	-	
33	040	space			97	140			
34	041	!	!		98	141	a	A	
35	042	"	"		99	142	b	B	
36	043	#	#		100	143	c	C	
37	044	$	$		101	144	d	D	
38	045	%	%		102	145	e	E	
39	046	&	&		103	146	f	F	
40	047	'	'		104	147	g	G	
41	050	(	(		105	150	h	H	
42	051	)	)		106	151	i	I	
43	052	*	*		107	152	j	J	
44	053	+	+		108	153	k	K	
45	054	,	,		109	154	l	L	
46	055	-	-		110	155	m	M	
47	056	.	.		111	156	n	N	
48	057				112	157	o	O	
49	060	0	0		113	160	p	P	
50	061	1	1		114	161	q	Q	
51	062	2	2		115	162	r	R	
52	063	3	3		116	163	s	S	
53	064	4	4		117	164	t	T	
54	065	5	5		118	165	u	U	
55	066	6	6		119	166	v	V	
56	067	7	7		120	167	w	W	
57	070	8	8		121	170	x	X	
58	071	9	9		122	171	y	Y	
59	072	:	:		123	172	z	Z	
60	073	;	;		124	173	{	{	
61	074	<	<		125	174	|	|	
62	075	=	=		126	175	}	}	
63	076	>	>		127	176			
64	077	?	?		128	177		rub out	delete

Appendix H—Machine-Dependent Routines

The following Fortran-callable routines are machine-dependent and must be provided locally. PASCAL type notation is used to indicate types. The type char indicates a Fortran integer whose value is interpreted as a Icon internal character. The type address indicates a Fortran integer array that is overlaid on other data such as a Fortran literal. The type boolean indicates a Fortran integer whose value is 0 or 1.

1. llc(c:char,a:array,i:integer):char — get the ith character character (zero-based) from the Fortran string literal at address a and return its ASCII code in c and as the function value. The value of i may be arbitrarily large (beyond the integer at a), but the Icon system assures that it is in the range of the literal. This routine has the sole responsibility of converting characters in Fortran literals into ASCII codes.

2. ldc(c:char,a:array,i:integer):char — get the ith character (zero-based) from the Icon string starting at address a and return it in c and as the function value. The value of i may be arbitrarily large (beyond the range of the integer at a), but the Icon system assures that it is in range of the string.

3. stc(c:char,a:address,i:integer):char — store character c at character location i (zero-based) from address a.

The routines ldc and stc are solely responsible for the movement of characters in Icon's internal character set. These characters always have 8 significant bits, regardless of the external character set. These routines embody the knowledge of the layout of characters within a Fortran integer. The machine-dependent parameter CHARSPERWORD in TDEF and IDEF must be set appropriately. It is *not* necessary that characters be stored in any particular format. For example, you may find it convenient not to use all the bits of your Fortran integer for storing characters if your word size is not evenly divisible by 8. For example, if the size of your Fortran integer is divisible by 9, you may wish to use 9 bits per character, ignoring the high-order bit. You should pack characters, however, to conserve storage space (i.e., it is not advisable to store only one character per Fortran integer).

4. tstb(a:address,i:integer);boolean — return the ith bit (zero-based) from address a. In this routine and the three routines that follow, the value of i may be arbitrarily large (beyond the integer at a). The value of bit i is not changed.

5. setb(a:address,i:integer);boolean — set the ith bit (zero-based) from address a to 1 and return the previous setting of this bit.

The two routines tstb and setb are solely responsible for the manipulation of bits in Fortran integers. The machine-dependent parameter WORDSIZE in TDEF and IDEF must be set appropriately. You do not need to actually use all the bits in you machine's word, but WORDSIZE must indicate how many bits are used and the routines above must correctly access the bits that are used.

6. syserr(s:string) — print the Fortran literal string s and terminate execution. The message should indicate that an error has occurred in the Icon system and the message should be printed to the user's standard output in a manner that is independent of Ratfor input/output. The string s is always terminated by a period.

7. runtim():integer — return elapsed CPU runtime for the Icon job in milliseconds. The Icon system expects a value measured from the beginning of execution and computes differences as necessary. This routine should *not* reset the time.

8. date4(date,year,time,secs:integer) — return date, year, time of day, and elasped seconds in date, year, time, and sec, respectively. The date is an integer in the form mmdd. For example, April 1 is 401. The year is a four-digit integer. The time is the current time of day in the form hhmm. For example, five minutes after noon is 1205. The seconds is an integer giving the elapsed seconds in the current minute (wall-clock time).

Appendix I—Icon Test Programs

program	data	function
hello	*none*	trivial test
fib	*none*	test of recursion and generators
comprs	*none*	test of scanning
scan	*none*	display of scanning
bridge	*none*	dealing bridge hands
kross	groups	word intersections
morsec	poem	Morse code translation
wordt	prog	word tabulation
recogn	senten	sentence recognition
graphm	graphs	graph manipulation
deriv	dexp	symbolic differentiation
rsg	gramm	random sentence generation

Appendix J—An Example of a ZZ Routine

```
include idef
##zz1(lab,n:integer) -- returns command line argument or fails (see Software Tools book).
# stable.
#
# n is number of arguments
# lab is used for generators and is not relevant here
# sp is the stack pointer; arguments consist of two words and are pushed on the stack,
# which grows downward
# signal indicates success or failure
#
subroutine zz1(lab, n)
   integer lab, n
   pointer ctos, p
   integer getarg
   include cutil
   include cmain
   include cmem

   if (n < 1)
      signal = 0                          # must have at least 1 argument
   else {
      sp = sp + 2*(n - 1)                 # remove trailing arguments
      call xderef                         # get the value of the argument
      call xcintg                         # make sure it's integer
      if (getarg(mem(mem(sp)), cbuf, MAXCHARS) ~= EOF) {
         p = ctos(cbuf)                   # getarg puts argument into cbuf, make
         mem(sp) = p                      # it an Icon string and return it
         }
      else
         signal = 0                       # argument not there
      }
   return
end
```