Version 8 of Icon*

Ralph E. Griswold

TR 90-1d

January 1, 1990; last revised April 3, 1990

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant CCR-8713690.

Version 8 of Icon

1. Introduction

The current version of Icon is Version 8. Version 8, which replaces Version 7.5, contains a number of features not present in earlier versions. The first edition of the Icon book [1] describes Version 5. The second edition of this book, which is in press, describes Version 8. This report serves as a temporary supplement to the first edition until the second edition is published. The descriptions here are brief and in some cases incomplete. Complete descriptions are contained in the second edition of the book.

Most of the language extensions in Version 8 are upward-compatible with previous versions of Icon and most programs written for earlier versions work properly under Version 8. However, some of the more implementation-dependent aspects described in the Version 5 book are now obsolete.

The major differences between Versions 5 and 8 are listed below. Features that are new since Version 7.5 are identified by squares (□):

A set data type.

□ Serial numbers for structures.

Functions for interfacing the operating system.

Functions for manipulating bits.

□ Functions for performing mathematical computations.

□ Keyboard functions.

□ Functions for getting variables from their names and vice versa.

Additional keywords related to co-expressions, csets, program location, storage management information, and language features.

□ Support for arithmetic on integers of arbitrarily large magnitude.

Declaration of procedures with a variable number of arguments.

The invocation of functions, procedures, and operators by their string names.

□ The invocation of functions and procedures with arguments from a list.

Syntactic support for the use of co-expressions in programmer-defined control operations.

Optional conversion of potential run-time errors to expression failure.

Additional options for sorting tables.

Co-expression tracing.

□ Interfaces for calling C functions from Icon and calling an Icon program from C.

Error trace back, which provides detailed information about the source of run-time errors.

Correction of the handling of scanning environments.

Correction of the handling of co-expression return points

□ Instrumentation of storage management.

A declaration to allow the inclusion of separately translated Icon programs.

2. New Language Features

2.1 Structures

Sets

Sets are unordered collections of values and have many of the properties normally associated with sets in the mathematical sense. The function

```
set(L)
```

creates a set that contains the distinct elements of the list L. For example,

```
set(["abc",3])
```

creates a set with two members, "abc" and 3.

The default value for an omitted argument to set() is an empty list. Consequently, set() creates an empty set.

Sets, like other data aggregates in Icon, need not be homogeneous — a set may contain members of different types. Sets can be members of sets, as in:

```
S1 := set([1,2,3])
S2 := set([S1,[]])
```

in which S2 contains two members, one of which is a set of three members and the other of which is an empty list.

Any specific value can occur only once in a set. For example,

```
set([1,2,3,3,1])
```

creates a set with the three members 1, 2, and 3. Set membership is determined the same way the equivalence of values is determined in the operation

```
x === y
```

For example,

```
set([[],[]])
```

creates a set that contains two distinct empty lists.

Several set operations are provided. The function member(S,x) succeeds and produces the value of x if x is a member of S, but fails otherwise. Note that

```
member(S1,member(S2,x))
```

succeeds if x is a member of both S1 and S2.

The function insert(S,x) inserts x into the set S and produces the value of S. Note that insert(S,x) is similar to put(L,x) in form. A set may contain (a pointer to) itself: insert(S,S) adds S as an member of itself. The function delete(S,x) deletes the member x from the set S and produces S.

The functions insert(S,x) and delete(S,x) always succeed, whether or not x is in S. This allows their use in loops in which failure may occur for other reasons. For example,

```
S := set()
while insert(S,read())
```

builds a set that consists of the (distinct) lines from the standard input file.

The operations

```
S1 ++ S2
S1 ** S2
S1 -- S2
```

create the union, intersection, and difference of S1 and S2, respectively. In each case, the result is a new set.

The use of these operations on csets is unchanged. There is no automatic type conversion between csets and sets; the result of the operation depends on the types of the arguments. For example,

```
'aeiou' ++ 'abcde'
```

produces the cset 'abcdeiou', while

```
set([1,2,3]) ++ set([2,3,4])
```

produces a set that contains 1, 2, 3, and 4. On the other hand,

```
set([1,2,3]) ++ 4
```

and

```
set(['a','b','c']) ++ 'd'
```

are erroneous.

The functions and operations of Icon that apply to other data aggregates apply to sets as well. For example, if S is a set, *S is the size of S (the number of members in it). Similarly, type(S) produces the string set.

The operation !S generates the members of S, but in no predictable order. Similarly, ?S produces a randomly selected member of S. These operations produce values, not variables — it is not possible to assign a value to !S or ?S.

The function copy(S) produces a new set, distinct from S, but which contains the same members as S. The copy is made in the same fashion as the copy of a list — the members themselves are not copied. The function sort(S) produces a list containing the members of S in sorted order. Sets occur after tables but before records in sorting.

Tables

The functions member(), insert(), and delete() apply to tables as well as sets.

The function member(T,x) succeeds if x is an entry value (key) in the table T, but fails otherwise.

The function insert(T,x,y) inserts the entry value x into table T with the assigned value y. If there already was an entry value x in T, its assigned value is changed. Note that insert has three arguments when used with tables, as compared to two when used with sets. An omitted third argument defaults to the null value.

The function delete(T,x) removes the entry value x and its corresponding assigned value from T. If x is not an entry value in T, no operation is performed; delete() succeeds in either case.

□ The function key(T) generates the keys in table T.

Sorting Order for Elements of Lists and Tables

A complete ordering is now defined for structures (lists, sets, tables, and records) and csets that appear in a larger structure to be sorted. Different types are still kept separate, but within each structure type, elements are now sorted chronologically (in the order they were created). All of the different record types are sorted together. Csets are sorted lexically, in the same fashion as strings.

Sorting Options for Tables

Two new options are available for sorting tables. These options are specified by the values 3 and 4 as the second argument of sort(T,i). Both of these options produce a single list in which the entry values and assigned values of table elements alternate. A value of 3 for i produces a list in which the entry values are in sorted order, and a value of 4 produces a list in which the assigned values are in sorted order.

The main advantage of the new sorting options is that they only produce a single list, rather than a list of lists as produced by the options 1 and 2. The amount of space needed for the single list is proportionally much less than for the list of lists.

□ **Serialization of Structures**

Structures are now serialized, starting at 1 with the first structure of each type. Serial numbers are shown in the string images of structures. For example, the value of image(set()) might be "set_5(0)".

2.2 Functions

Most of the new functions are described in this section. See other sections for functions related to specific features. *Note:* Some implementations have additional functions. These are described in user manuals.

System-Interface Functions

getenv(s)

getenv(s) produces the value of the environment variable s. It fails if the environment variable is not set. It also fails on systems that do not support environment variables.

remove(s)

remove(s) removes the file named s. Subsequent attempts to open the file fail, unless it is created anew. If the file is open, the behavior of remove(s) is system dependent. remove(s) fails if it is unsuccessful.

rename(s1,s2)

rename(s1,s2) causes the file named s1 to be henceforth known by the name s2. The file named s1 is effectively removed. If a file named s2 exists prior to the renaming, the behavior is system dependent. rename(s1,s2) fails if unsuccessful, in which case if the file existed previously it is still known by its original name. Among the other possible causes of failure would be a file currently open, or a necessity to copy the file's contents to rename it.

seek(f,i)

seek(f,i) seeks to position i in file f. As with other positions in Icon, a nonpositive value of i can be used to reference a position relative to the end of f. i defaults to 1. seek(f,i) produces f but fails if an error occurs.

where(f)

where(f) produces the current byte position in the file f.

Characters and Ordinals

char(i)

char(i) produces a string containing the character whose internal representation, or ordinal, is the integer i. i must be between 0 and 255 inclusive. If i is out of range, or not convertible to integer, a run-time error occurs.

ord(s)

ord(s) produces an integer between 0 and 255 representing the ordinal, or internal representation, of a character. If s is not convertible to string, or if the string's length is not 1, a run-time error occurs.

Tab Expansion and Insertion

detab(s,i1,i2,...,in)

detab(s,i1,i2,...,in) replaces each tab character in s by one or more space characters, using tab stops at i1,i2...,in, and then additional tab stops created by repeating the last interval as necessary. The default is detab(s,9).

Tab stops must be positive and strictly increasing. There is an implicit tab stop at position 1 to establish the first interval. Examples are:

detab(s)	tab stops at 9, 17, 25, 33, ...
detab(s,5)	tab stops at 5, 9, 13, 17, ...
detab(s,8,12)	tab stops at 8, 12, 16, 20, ...
detab(s,11,18,30,36)	tab stops at 11, 18, 30, 36, 42, 48, ...

For purposes of tab processing, "\b" has a width of –1, and "\r" and "\n" restart the counting of positions. Other non-printing characters have zero width, and printing characters have a width of 1.

entab(s,i1,i2,...,in)

entab(s,i1,i2,...,in) replaces runs of consecutive spaces with tab characters. Tab stops are specified in the same manner as for the detab function. Any existing tab characters in s are preserved, and other nonprinting characters are treated identically with the detab function. The default is entab(s,9).

A lone space is never replaced by a tab character; however, a tab character may replace a single space character that is part of a longer run.

Bit-Wise Functions

The following functions operate on the individual bits composing one or two integers. All produce an integer result.

iand(i,j)	bit-wise *and* of i and j
ior(i,j)	bit-wise inclusive *or* of i and j
ixor(i,j)	bit-wise exclusive *or* of i and j
icom(i)	bit-wise complement (one's complement) of i
ishift(i,j)	If j is positive, i shifted left by j bit positions. If j is negative, i shifted right by –j bit positions. In all cases, vacated bit positions are filled with zeroes.

□ **Math Functions**

The following functions are provided for performing mathematical computations:

sin(r)	sine of r
cos(r)	cosine of r
tan(r)	tangent of r
asin(r)	arc sine of r
acos(r)	arc cosine of r
atan(r1,r2)	arc tangent of r1 / r2
dtor(r)	radian equivalent of r given in degrees
rtod(r)	degree equivalent of r given in radians
sqrt(r)	square root of r
exp(r)	*e* raised to the power r
log(r1,r2)	logarithm of r1 to the base r2

□ **Keyboard Functions**

The following functions for keyboard input and output are available on systems that support such capabilities:

getch()

getch() waits until a character has been entered from the keyboard and then produces the corresponding one-character string. The character is not displayed.

getche()

getche() waits until a character has been entered from the keyboard and then produces the corresponding one-character string. The character is displayed.

kbhit()

kbhit() succeeds if a character is available for getch() or getche() but fails otherwise.

□ **Variables and Names**

name(v)

name(v) produces the string name of the variable v. The string name of an identifier or keyword is just as it appears in the program. The string name of a subscripted string-valued variable consists of the variable and the subscript, as in "line[2+:3]". The string name of a list or table reference consists of the data type and the subscripting expression, as in "list[3]". The string name of a record field reference consists of the record type and field name with a separating period, as in "complex.r".

variable(s)

variable(s) produces the variable for the identifier or keyword with the name s.

Executable Images

save(s)

The function save(s) saves an executable image of an executing Icon program in the file named s. This function presently is implemented only on BSD UNIX systems. See Section 2.10 for a method of determining if this feature is implemented.

save() can be called from any point in a program. It accepts a single argument that names the file that is to receive the resulting executable. The named file is created if it does not exist. Any output problems on the file cause save() to fail. For lack of anything better, save() produces the number of bytes in the data region.

When the new executable is run, execution of the Icon program begins in the main procedure. Global and static variables have the value they had when save() was called, but all dynamic local variables have the null value. Any initial clauses that have been executed are not re-executed. As usual, arguments present on the command line are passed to the main procedure as a list. Command line input and output redirections are processed normally, but any files that were open are no longer open and attempts to read or write them will fail.

When the Icon interpreter starts up, it examines a number of environment variables to determine various operational parameters. When a saved executable starts up, the environment variables are not examined; the parameter values recorded in the executable are used instead. Note that many of the parameter values are dynamic and may have changed considerably from values supplied initially.

Consider an example:

```
global hello
procedure main()
   initial {
      hello := "Hello World!"
      save("hello") | stop("Error saving to 'hello'")
      exit()
      }
   write(hello)
end
```

The global variable hello is assigned "Hello World!" and then the interpreter is saved to the file hello. The program then exits. When hello is run, main's initial clause is skipped since it has already been executed. The variable hello has retained its value and the call to write produces the expected greeting.

It is possible to call save() any number of times during the course of execution. Saving to the same file each time is rather uninteresting, but imagine a complex data structure that passes through levels of refinement and then saving out a series of executables that capture the state of the structure at given times.

Saved executables contain the entire data space present at the time of the save and thus can be quite large. Saved executables on a VAX are typically around 250k bytes.

□ **Large Integers**

Integers now are not limited in magnitude by machine architecture. This feature is not supported on all implementations of Icon because it increases the size of the Icon system by a significant amount. See Section 2.10 for a method of determining if this feature is implemented.

Negative large integers become positive if shifted right by ishift().

The string image of a large integer whose decimal representation would have more than about 25 digits has the form integer(~*n*), where *n* is the approximate number of decimal digits.

Integer Sequences

seq(i,j) generates an infinite sequence of integers starting at i with increments of j. An omitted or null-valued argument defaults to 1. For example, seq() generates 1, 2, 3, … .

2.3 Procedures and Functions

Procedures with a Variable Number of Arguments

A procedure can be made to accept a variable number of arguments by appending [] to the last (or only) parameter in the parameter list. An example is:

```
procedure p(a,b,c[])
   suspend a + b + !c
end
```

If called as p(1,2,3,4,5), the parameters have the following values:

```
a       1
b       2
c       [3,4,5]
```

The last parameter always contains a list. This list consists of the arguments not used by the previous parameters. If the previous parameters use up all the arguments, the list is empty. If there are not enough arguments to satisfy the previous parameters, the null value will be used for the remaining ones, but the last parameter will still contain the empty list.

□ **Invocation with a List of Values**

The operation p!L invokes p with the arguments in the list L. For example,

```
write![1,2,3]
```

is equivalent to

```
write(1,2,3)
```

Note that this feature allows functions and procedures to be invoked with a number of arguments that need not be known when the program is written.

Invocation by String Name

A string-valued expression that corresponds to the name of a procedure or operation can be used in place of the procedure or operation in an invocation expression. For example,

```
"image"(x)
```

produces the same call as

```
image(x)
```

and

```
"–"(i,j)
```

is equivalent to

```
i – j
```

In the case of operator symbols with unary and binary forms, the number of arguments determines the operation. Thus

```
"–"(i)
```

is equivalent to

```
–i
```

Since to-by is an operation, despite its reserved-word syntax, it is included in this facility with the string name "..." . Thus

```
"..."(1,10,2)
```

is equivalent to

```
1 to 10 by 2
```

□ Similarly, range specifications are represented by "[:]", so that

```
"[:]"(s,i,j)
```

is equivalent to

```
s[i:j]
```

The subscripting operation is available with the string name "[]". Thus

```
"[]"(&lcase,3)
```

produces c.

Defaults are not provided for omitted or null-valued arguments in this facility. Consequently,

```
"..."(1,10)
```

results in a run-time error when it is evaluated.

Arguments to operators invoked by string names are dereferenced. Consequently, string invocation for assignment operations is ineffective and results in error termination.

String names are available for the operations in Icon, but not for control structures. Thus

"|"(*expr*$_1$,*expr*$_2$)

is erroneous. Note that string scanning is a control structure.

Field references, of the form

expr . fieldname

are not operations in the ordinary sense and are not available via string invocation. In addition, conjunction is not available via string invocation, since no operation is actually performed.

String names for procedures are available through global identifiers. Note that the names of functions, such as image, are global identifiers. Similarly, any procedure-valued global identifier may be used as the string name of a procedure. Thus, in

```
global q

procedure main()
   q := p
   "q"("hi")
end

procedure p(s)
   write(s)
end
```

the procedure p is invoked via the global identifier q.

The function proc(x,i) converts x to a procedure, if possible. If x is procedure-valued, its value is returned unchanged. If the value of x is a string that corresponds to the name of a procedure as described previously, the corresponding procedure value is returned. The value of i is used to distinguish between unary and binary operators. For example, proc("∗",2) produces the multiplication operator, while proc("∗",1) produces the size operator. The default value for i is 1. If x cannot be converted to a procedure, proc(x,i) fails.

2.4 String Scanning

Scanning environments (&subject and &pos) are now maintained correctly. In particular, they are are now restored when a scanning expression is exited via break, next, return, fail, and suspend. This really is a correction of a bug, although the former handling of scanning environments is described as a ''feature'' in the first edition of the Icon book. Note also that this change could affect the behavior of existing Icon programs.

2.5 Co-Expressions

Co-expression return points are now handled properly, so that co-expressions can be used as coroutines.

The image of a co-expression now includes a serial number. Numbers start with 1 for &main and increase as new co-expressions are created.

Co-expression activation and return are now traced along with procedure calls and returns if &trace is nonzero. Co-expression tracing shows the identifying numbers.

The value of &current is the current co-expression.

The initial size of &main is now 1, instead of 0, to reflect its activation to start program execution.

An attempt to refresh &main results in a run-time error. (Formerly, it caused a memory violation.)

2.6 Programmer-Defined Control Operations

Co-expressions can be used to provide programmer-defined control operations. This facility uses an alternative syntax for procedure invocation in which the arguments are passed in a list of co-expressions. This syntax uses braces in place of parentheses:

p{*expr*$_1$, *expr*$_2$, …, *expr*$_n$}

is equivalent to

p([create *expr*$_1$, create *expr*$_2$, …, create *expr*$_n$])

Note that

p{}

is equivalent to

```
p([])
```

2.7 Input and Output

There no longer are any length limitations on the string produced by read() or reads(), nor on the length of the argument to system() or the first argument of open().

Formerly the value returned by write(x1,x2,...,xn) and writes(x1,x2,...,xn) was the last written argument *converted to a string*. The conversion is no longer performed. For example, the value returned by write(1) is the integer 1.

Errors during writing (such as inadequate space on the output device) now cause error termination.

The function read(f) reads the last line of the file f, even if that line does not end with a newline (Version 5 discarded such a line).

If the file f is open as a pipe, close(f) produces the system code resulting from closing f instead of f.

Icon no longer performs its own I/O buffering; instead, this function is left to the operating system on which Icon runs. The environment variable NBUFS is no longer supported.

2.8 Errors

Error Trace Back

A run-time error now shows a trace back, giving the sequence of procedure calls to the site of the error, followed by a symbolic rendering of the offending expression. For example, suppose the following program is contained in the file max.icn:

```
procedure main()
   i := max("a",1)
end

procedure max(i,j)
   if i > j then i else j
end
```

Its execution in Version 8 produces the following output:

```
Run–time error 102
File max.icn; Line 6
numeric expected
offending value: "a"
Trace back:
   main()
   max("a",1) from line 2 in max.icn
   {"a" > 1} from line 6 in max.icn
```

Error Conversion

The keyword &error controls the conversion of potential run-time errors into expression failure. It behaves like &trace: if it is zero, the default value, errors are handled as usual. If it is non-zero, errors are treated as failure and &error is decremented.

There are a few errors that cannot be converted to failure: arithmetic overflow and underflow, stack overflow, and errors during program initialization.

When an error is converted to failure in this way, three keywords are set:

&errornumber is the number of the error (e.g., 101). Reference to &errornumber fails if there has not been an error.

&errortext is the error message (e.g., integer expected).

&errorvalue is the offending value. Reference to &errorvalue fails if there is no specific offending value.

The function errorclear() removes the indication of the last error. Subsequent references to &errornumber fail until another error occurs.

The function runerr(i,x) causes program execution to terminate with error number i as if a corresponding run-time error had occurred. If i is the number of a standard run-time error, the corresponding error text is printed; otherwise no error text is printed. The value of x is given as the offending value. If x is omitted, no offending value is printed.

This function is provided so that library procedures can be written to terminate in the same fashion as built-in operations. It is advisable to use error numbers for programmer-defined errors that are well outside the range of numbers used by Icon itself. See Appendix B. Error number 500 has the predefined text "program malfunction" for use with runerr(). This number is not used by Icon itself.

A call of runerr() is subject to conversion to failure like any other run-time error.

2.9 □ Icon-C Interfaces

C functions now can be called from Icon programs [2]. The function callout(x,x1,x2,...,xn) calls the C function designated by x and passes it the arguments x1, x2, ..., xn. The method of designating C functions and passing arguments is system-dependent.

An Icon program also can be called from C. The method of doing this is described in [2].

2.10 Implementation Features

The &features generates the features of the implementation on which the current program is running. For example, on a BSD UNIX implementation,

```
every write(&features)
```

produces

```
UNIX
ASCII
calling to Icon
co–expressions
direct execution
environment variables
error trace back
executable images
expandable regions
external functions
large integers
math functions
memory monitoring
pipes
string invocation
system function
```

Similarly, a program that uses co-expressions can check for the presence of this feature:

```
if not(&features == "co–expressions") then runerr(401)
```

2.11 Storage Management

Storage is allocated automatically during the execution of an Icon program, and garbage collections are performed automatically to reclaim storage for subsequent reallocation. There are three storage regions: static, string, and block. Only implementations in which regions can be expanded support a static region. See [3] for more information.

An Icon programmer normally need not worry about storage management. However, in applications that require a large amount of storage or that must operate in a limited amount of memory, some knowledge of the storage management process may be useful.

The keyword &collections *generates* four values associated with garbage collection: the total number since program initiation, the number triggered by static allocation, the number triggered by string allocation, and the number triggered by block allocation. The keyword &regions *generates* the current sizes of the static, string, and block regions. The keyword &storage *generates* the current amount of space used in the static, string, and block regions. The value given for the static region presently is not meaningful.

Garbage collection is forced by the function collect(i,j). The value of i specifies the region and the value of j specifies the amount of space that must be free following the collection. The regions are designated as follows:

i	region
1	static
2	string
3	block

The region specified is reflected in the values generated by &collections. A value of 0 for i causes a garbage collection without a specific region. In this case, the value of j is irrelevant. The default values for i and j are 0.

2.12 □ Memory Monitoring

Storage allocation and garbage collection now are instrumented [4]. Normally, this instrumentation is disabled. It is enabled by setting the environment variable MEMMON to the name of an allocation history file to receive memory-monitoring data.

There are several tools for processing memory-monitoring data, including ones for producing interactive visualizations of storage management. See [4] for more information.

The function mmpause(s) causes a pause in interactive visualizations, displaying the identification s. The default for s is "programmed pause".

The function mmshow(x,s) redraws the object x in a color specified by s. The color specifications are:

"b"	black
"g"	gray
"w"	white
"h"	highlight: blinking black and white
"r"	redraw in normal color

The default is "r".

The function mmout(s) writes s (without interpretation) as a separate line to the allocation history file.

2.13 Odds and Ends

Other Keywords

In addition to the new keywords mentioned above, there are four others:

&digits, whose value is '0123456789'.

□ &letters, whose value is a cset containing the 52 upper- and lowercase letters.

&line, the current source-code line number.

&file, the current source-code file name.

Addition to suspend

The suspend control structure now has an optional do clause, analogous to every-do. If a do clause is present in a suspend control structure, its argument is evaluated after the suspend is resumed and before possible suspension with another result.

For example, the following expression might be used to count the number of suspensions:

```
suspend expr do count +:= 1
```

String and Cset Images

''Unprintable'' characters in strings and csets now are imaged with hexadecimal escape sequences rather than with octal ones.

Display Output

The function display(f,i) now prints the image of the current co-expression before listing the values of variables.

Tracing

If the value of &trace is negative, it is decremented every time a trace message is written. Previously it was left unchanged. This change does not affect tracing itself, but it does allow the number of trace messages that have been written to be determined by a running program.

□ **Reserved Words**

The reserve word dynamic has been deleted.

□ **Character Equivalents**

The character pairs $(, $), $<, and $> are equivalent to {, }, [, and], respectively. These equivalents are provided for use on EBCDIC systems that cannot input and output braces and brackets, but the equivalents also can be used on ASCII systems.

□ **Numeric Conversion**

If large integers are not supported, an attempt to convert a string of digits whose numeric value would be too large for an Icon integer now fails instead of producing a real number.

3. Running Icon

3.1 Command-Line Options

Options to icont must precede file names on the command line. For example,

```
icont –o manager proto.icn
```

not

```
icont proto.icn –o manager
```

The position of options has always been documented this way, but it was not enforced previously. Consequently, options in the incorrect position that worked before will not work now. This problem is most likely to occur in scripts that were composed under earlier versions of Icon.

Note that the –x option is an exception; it must occur after all file names for icont, since any command-line arguments after –x apply to execution (iconx).

3.2 The Link Declaration

The link declaration simplifies the inclusion of separately translated libraries of Icon procedures. If icont [5] is run with the –c option, source files are translated into intermediate *ucode* files (with names ending in .u1 and .u2). For example,

```
icont –c libe.icn
```

produces the ucode files libe.u1 and libe.u2. The ucode files can be incorporated in another program with the new link declaration, which has the form

```
link libe
```

The argument of link is, in general, a list of identifiers or string literals that specify the names of files to be linked (without the .u1 or .u2). Thus, when running under UNIX,

```
link libe, "/usr/icon/ilib/collate"
```

specifies the linking of libe in the current directory and collate in /usr/icon/ilib. The syntax for paths may be different for other operating systems.

The environment variable IPATH controls the location of files specified in link declarations. The value of IPATH should be a blank-separated string of the form p_1 p_2 … p_n where each p_i names a directory. Each directory is searched in turn to locate files named in link declarations. The default value of IPATH is the current directory. The current directory is always searched first, regardless of the value of IPATH.

Linker Options

The option to generate diagnostic (.ux) files during linking has been changed from –D to –L.

The amount of space needed to associate source-program line numbers with executable code can be set by –Sn*n*. The default value of *n* is 1000. Similarly, the amount of space needed to associate file names with executable code can be set by –SF*n*. The default is 10.

□ **Path to** iconx

For implementations that support direct execution of icode files, the hardwired path to iconx is overridden by the value of the environment variable ICONX, if it is set. If ICONX is not set and iconx is not found on the hardwired path, PATH is searched for it.

Block Region Size

The environment variables BLOCKSIZE and BLKSIZE now are synonyms for HEAPSIZE.

File Names

During translation and linking, the suffix .u is interpreted as .u1, and no suffix is interpreted as .icn.

On some systems, the icode file produced by the linker has the extension icx. For example, on MS-DOS,

```
icont prog.icn
```

produces an icode file named prog.icx. The extension need not be given when using iconx, as in:

```
iconx prog
```

Redirection of Error Output

The option –e now allows standard error output to be redirected to a file. For example,

```
iconx –e prog.err prog
```

executes prog and sends any error output to the file prog.err. If – is given in place of a file name, error output is redirected to standard output. On systems on which standard output can be redirected to a file, –e – causes both error output and standard output go to that file. For example,

```
iconx –e – prog >prog.out
```

redirects both error output and standard output to prog.out.

Version Checking

The Icon translator converts a source-language program to an intermediate form, called *ucode*. The Icon linker converts one or more ucode files to a binary form called *icode*. The format of Version 8 ucode and icode files is different from that of earlier versions. To avoid the possibility of malfunction due to incompatible ucode and icode formats, Version 8 checks both ucode and icode files and terminates processing with an error message if the

versions are not correct.

□ **Program Location Information**

A comment that begins at the beginning of a line and has the form

#line *n* "*f*"

changes the current source-program line number and file name used by the Icon translator to *n* and *f, respectively*.

□ **Warning Messages**

The Icon translator now issues a warning message if the dereferencing operator is applied to a numeric literal, as in .25.

Miscellaneous

Some run-time environment variables have changed; see Appendix A. Several error messages have been changed. Appendix B contains a list of run-time error messages for Version 8.

4. Effects of Implementation Changes

There are many differences between the implementation of Version 8 of Icon and earlier versions. Most of these changes only affect performance and are otherwise invisible to users.

Changes in the techniques used for hashing, however, change the order in which elements are generated from sets and tables and the random selection of elements of sets and tables.

In addition, the order of generation and random selection from sets and tables may vary between implementations.

5. Obsolete and Changed Features

The original implementation of Version 5 supported both a compiler (iconc) and an interpreter (icont). Version 8 supports only an interpreter. It is not possible to load C functions with the interpreter as it was with the compiler. A system for personalized interpreters [6] is included with Version 8 for UNIX systems to make it comparatively easy to add new functions and otherwise modify the Icon run-time system.

The reserved word dynamic is no longer available; local is equivalent.

6. Bugs and Problems

- Line numbers sometimes are wrong in diagnostic messages related to lines with continued quoted literals.
- Large-integer arithmetic is not supported in i to j and seq(). Large integers cannot be assigned to keywords.
- Large-integer literals are constructed at run-time. Consequently, they should not be used in loops where they would be constructed repeatedly.
- Conversion of a large integer to a string is quadratic in the length of the integer. Conversion of very a large integer to a string may take a very long time and give the appearance of an endless loop.
- Integer overflow on exponentiation may not be detected during execution. Such overflow may occur during type conversion.
- In some cases, trace messages may show the return of subscripted values, such as &null[2], that would be erroneous if they were dereferenced.

- If a long file name for an Icon source-language program is truncated by the operating system, mysterious diagnostic messages may occur during linking.
- Stack overflow is checked using a heuristic that may not always be effective.
- If an expression such as

  ```
  x := create expr
  ```

  is used in a loop, and x is not a global variable, unreferenceable co-expressions are generated by each successive create operation. These co-expressions are not garbage collected. This problem can be circumvented by making x a global variable or by assigning a value to x before the create operation, as in

  ```
  x := &null
  x := create expr
  ```

- Stack overflow in a co-expression may not be detected and may cause mysterious program malfunction.

7. Possible Differences Among Version 8 Implementations

A few aspects of the implementation of Version 8 are specific to different computer architectures and operating systems. Co-expressions require a context switch that is implemented in assembly language. If this context switch is not implemented, an attempt to activate a co-expression results in error termination.

Some features of Icon, such as opening a pipe for I/O and the system() function, are not supported on all operating systems. See specific user manuals for details.

Acknowledgements

Mark Emmer, Clint Jeffery, Sandra Miller, Chris Smith, Gregg Townsend, and Ken Walker contributed to Version 8 of Icon.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. R. E. Griswold, *Icon-C Calling Interfaces*, The Univ. of Arizona Tech. Rep. 90-8, 1990.
3. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
4. G. M. Townsend, *The Icon Memory Monitoring System*, The Univ. of Arizona Icon Project Document IPD113, 1990.
5. R. E. Griswold, *ICONT(1)*, manual page for *UNIX Programmer's Manual*, The Univ. of Arizona Icon Project Document IPD109, 1990.
6. R. E. Griswold, *Personalized Interpreters for Version 8 of Icon*, The Univ. of Arizona Tech. Rep. 90-3, 1990.

Appendix A — Environment Variables

There are a number of environment variables that can be set to override the default values for sizes of Icon's storage regions and other run-time parameters. Except for ICONX, NOERRBUF, and ICONCORE, the values assigned to these environment variables must be numbers. Default values for regions vary from system to system and are given in user manuals. Some implementations also have other environment variables.

The environment variables are:

ICONX	If set, this environment variable specifies the location of iconx used to execute an icode file.
TRACE	Specifies the initial value of &trace. The default value is zero.
NOERRBUF	If set, standard error output is not buffered.
ICONCORE	If set, a core dump is produced in the case of error termination.
MSTKSIZE	Specifies the size in words of the main interpreter stack.
STRSIZE	Specifies the initial size in bytes of the allocated string region.
BLOCKSIZE	Specifies the initial size in bytes of the allocated block region. HEAPSIZE and BLKSIZE are synonyms for BLOCKSIZE.
STATSIZE	Specifies the initial size in bytes of the static region in which co-expressions are allocated.
STATINCR	Specifies the increment for expanding the static region. The default increment is one-fourth the initial size of the static region.
COEXPSIZE	Specifies the size in words of co-expression blocks.
QLSIZE	Specifies the amount of space used for pointers to strings during garbage collection. Used only on implementations with fixed memory regions.
MEMMON	Name of the file for memory-monitoring data.

An inappropriate setting of an environment variable prevents the program from running.

Appendix B — Run-Time Error Messages

A list of run-time error numbers and corresponding messages follows. Some implementations have additional run-time errors.

101	integer expected
102	numeric expected
103	string expected
104	cset expected
105	file expected
106	procedure or integer expected
107	record expected
108	list expected
109	string or file expected
110	string or list expected
111	variable expected
112	invalid type to size operation
113	invalid type to random operation
114	invalid type to subscript operation
115	list, set, or table expected
116	invalid type to element generator
117	missing main procedure
118	co-expression expected
119	set expected
120	cset or set expected
121	function not supported
122	set or table expected
123	invalid type
124	table expected
201	division by zero
202	remaindering by zero
203	integer overflow
204	real overflow, underflow, or division by zero
205	value out of range
206	negative first operand to real exponentiation
207	invalid field name
208	second and third arguments to map of unequal length
209	invalid second argument to open
210	non-ascending arguments to detab/entab
211	by value equal to zero
212	attempt to read file not open for reading
213	attempt to write file not open for writing
214	input/output error
215	attempt to refresh &main
216	external function not found

301	evaluation stack overflow
302	system stack overflow
303	inadequate space for evaluation stack
304	inadequate space in qualifier list
305	inadequate space for static allocation
306	inadequate space in string region
307	inadequate space in block region
308	system stack overflow in co-expression
401	co-expressions not implemented
500	program malfunction