Version 6 of Icon*

Ralph E. Griswold
William H. Mitchell
Janalee O'Bagy

TR 86-10b

June 8, 1986; Last Revision September 24, 1986

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8401831.

Version 6 of Icon

1. Background

Different versions of the Icon programming language are identified by numbers. The first, Version 1, was superceded by Version 2. Starting with Version 3, the implementation of Icon was completely changed. While Version 2 is still in use on some computers, Version 5 is by far the mostly widely used version of Icon. The features of Version 5 are described in the "Icon book" [1]. Since this book is the only complete and generally available description of a widely used version of Icon, Version 5 is sometimes called "standard Icon".

Since Icon is the byproduct of a research effort that is concerned with the development of novel programming language features, some extensions to the standard language are inevitable. These extensions are incorporated as features of new releases of the implementation. These releases are identified by minor version numbers that are separated from the major version number by a decimal point. For example, Version 5.10 is the tenth minor revision of Version 5.

Because of the relationship between language design and implementation, major version numbers usually contain significant language changes that have accumulated over a number of minor revisions, while minor version numbers primarily reflect implementation changes.

Recently an implementation of Icon written almost entirely in C has been completed. This implementation is so different from previous ones that some designation is needed to distinguish it from other implementations. It therefore has been designated Version 6. Despite the change in major version number, the Version 6 language is nearly the same as the Version 5.10 language and the Version 5 book continues to serve as the primary reference for Version 6. Since a revision of a book is a major and time-consuming process, this report is provided to supplement the present Icon book. Together, the book and this report provide a description of Version 6.

Most of the language extensions in Version 6 are upward-compatible with Version 5 and almost all programs that contain only standard Version 5 features work properly under Version 6. However, some of the more implementation-dependent aspects described in the Version 5 book are now obsolete and there are significant new language features.

2. Features of Version 6

Version 6 extensions consist of a declaration to facilitate the use of Icon procedure libraries, a new set data type, new options for sorting tables, new syntax to support the use of co-expressions in programmer-defined control operations, the invocation of functions and operators by their string names, and a few new functions.

2.1 The Link Declaration

The link declaration simplifies the inclusion of separately translated libraries of Icon procedures. If icont [2] 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*,

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

```
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[1] 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.

2.2 Sets

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

```
set(a)
```

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

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

creates a set with two members, abc and 3. Note that

```
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, like other Icon data aggregates, are represented by pointers to the actual data. 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 returns 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 returns the value of s. Note that insert(s,x) is similar to put(a,x) in form. A set may contain (a pointer to) itself:

[1]The separator is a colon for UNIX systems under Versions 5.9 and 5.10.

```
insert(s, s)
```

adds s as an member of itself.

The function

```
delete(s, x)
```

deletes the member x from the set s and returns the value of 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
```

results in Run-time Error 119 (set expected).

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 string images of sets are in the same style as for other aggregates, with the size enclosed in parentheses. Therefore,

```
s := set(["abc", 3])
write(image(s))
```

writes set(2).

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 Icon's collating sequence.

Examples

Word Counting:

The following program lists, in alphabetical order, all the different words that occur in standard input:

```
procedure main()
   letter := &lcase ++ &ucase
   words := set([])
   while text := read() do
      text ? while tab(upto(letter)) do
         insert(words, tab(many(letter)))
   every write(!sort(words))
end
```

The Sieve of Eratosthenes:

The follow program produces prime numbers, using the classical "Sieve of Eratosthenes":

```
procedure main()
   local limit, s, i
   limit := 5000
   s := set([])
   every insert(s, 1 to limit)
   every member(s, i := 2 to limit) do
      every delete(s, i + i to limit by i)
   primes := sort(s)
   write("There are ", *primes, " primes in the first ", limit, " integers.")
   write("The primes are:")
   every write(right(!primes, *limit + 1))
end
```

2.3 Sorting 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. For example, if the table wcount contains elements whose entry values are words and whose corresponding assigned values are counts, the following code segment writes out a list of the words and their counts, with the words in alphabetical order:

```
a := sort(wcount, 3)
every i := 1 to *a - 1 by 2 do
   write(a[i], " : ", a[i + 1])
```

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.

2.4 Programmer-Defined Control Operations

As described in [3], co-expressions can be used to provide programmer-defined control operations. Version 6 provides support for this facility by means of 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{expr1, expr2, ..., exprn}
```

is equivalent to

```
p([create expr1, create expr2, ..., create exprn])
```

Note that

```
p{}
```

is equivalent to

```
p([])
```

2.5 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]
```

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.

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

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

produces c.

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.

2.6 New Functions

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 in the preceding section, 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.

The function 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, ...

Storage is allocated automatically during the execution of an Icon program, and garbage collections are performed automatically to reclaim storage for subsequent reallocation. Garbage collection normally only occurs when it is necessary. Garbage collection can be forced by the function collect().

2.7 Minor Language Changes

There are two minor language changes:

- The keyword &options of Version 5.10 is not present in Version 6.
- Version 6 reads the last line of a file, even if that line does not end with a newline; Version 5 discards such a line.

2.8 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 6 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 6 checks both ucode and icode files and terminates processing with an error message if the versions are not correct.

3. Obsolete and Changed Features of Version 5

The original implementation of Version 5 supported both a compiler (iconc) and an interpreter (icont). Version 6 supports only an interpreter. Interpretation is only slightly slower than the execution of compiled code and the interpreter is portable and gets into execution much more quickly than the compiler. However, it is not possible to load C functions with the interpreter as it was with the compiler. A system for personalized interpreters [4] is included with Version 6 for UNIX systems to make it comparatively easy to add new functions and otherwise modify the Icon run-time system.

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

4. Known Bugs in Version 6

- The text of some translator error messages is incorrect and may not correctly reflect the nature of errors.
- The translator does not detect arithmetic overflow in conversion of numeric literals. Very large numeric literals may have incorrect values.
- Integer overflow on multiplication and exponentiation is not detected during execution. Such overflow may occur during type conversion.
- Line numbers may be wrong in diagnostic messages related to lines with continued quoted literals.
- 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.
- Co-expressions were designed to support coroutine programming as well as the more usual application for the controlled production of the results of a generator. However, the implementation is not sufficiently general to support coroutines. In Version 5, the use of co-expressions in a coroutine fashion could produce program malfunction. In Version 6, such uses are prevented and cause error termination. As a

consequence, some uses of co-expressions that worked in Version 5 may not work in Version 6. In particular, the example in Section 13.4.2 of the Icon book does not work in Version 6.

- The garbage collector was designed for machines with comparatively small address spaces and may not perform well for computers with very large address spaces. In particular, if an attempt is made to create a very large data object that will not fit into memory (such as a million-element list), it takes an inordinately long time to determine that the object can not be allocated.

5. Possible Differences Among Version 6 Implementations

Version 6 of Icon is written almost entirely in C and most of its features are machine-independent. Appendix B of the Icon book lists differences that may occur because of different machine architectures. In addition to the differences listed there, the implementation of sets and tables uses a parameter that is different for 16- and 32-bit computers. This parameter determines how set members and table elements are physically stored as a result of hashing. The difference only is noticeable in the results produced by !x and ?x, where x is a set or a table.

A few aspects of the implementation of Version 6 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. Arithmetic overflow checking also generally requires assembly-language routines and may not be supported on some implementations of Version 6.

Some features of Icon, such as opening a pipe for i/o and the system function, are closely related to UNIX. These features should work correctly for Version 6 running on UNIX systems, but they may not be supported on other operating systems.

Acknowledgements

In addition to the authors of this report, several persons contributed to the implementation of Version 6 of Icon. The implementation of sets and the new sorting options were done by Rob McConeghy. Other major contributors include Gregg Townsend, Chris Janton, and Kelvin Nilsen.

References

1. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.

2. Griswold, Ralph E. *ICONT(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona. May 1986.

3. Griswold, Ralph E. and Michael Novak. "Programmer-Defined Control Operations", *The Computer Journal*, Vol. 26, No. 2 (May 1983).

4. Griswold, Ralph E. *Personalized Interpreters for Version 6.0 of Icon*, Technical Report TR 86-12, Department of Computer Science, The University of Arizona. May 1985.

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. The values assigned to these environment variables should be numbers. Care should be taken in assigning values to region sizes; unreasonable values may cause Icon to malfunction.

The environment variables are:

TRACE	The initial value of &trace. The default value is zero.
NBUFS	The number of input-output buffers. The default value is 10 for computers with large address spaces and 5 for computers with small address spaces.
NOERRBUF	No buffering of standard error output.
ICONCORE	Produce a core dump in the case of error termination.
MSTKSIZE	The size in words of the main interpreter stack. The default value is 10,000 for machines with large address spaces and 3000 for machines with small address spaces.
STRSIZE	The initial size in bytes of the allocated string region. The default value is 51,200 for machines with large address spaces and 10,240 for machines with small address spaces.
HEAPSIZE	The initial size in bytes of the allocated block region. The default value is 51,200 for machines with large address spaces and 10,240 for machines with small address spaces.
STATSIZE	The initial size in bytes of the static region in which co-expressions are allocated. The default value is 20,480 for machines with large address spaces and 4,096 for machines with small address spaces.
STATINCR	The increment for expanding the static region. The default increment is one-fourth the initial size of the static region.
COEXPSIZE	The size in words of co-expression blocks. The default value is 2,000 for machines with large address spaces and 1,000 for machines with small address spaces.

Appendix B — Run-Time Error Messages

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 or table expected
116	invalid type to element generator
117	missing main procedure
118	co-expression expected
119	set 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	argument to system function too long
211	by clause equal to zero
212	attempt to read file not open for reading
213	attempt to write file not open for writing
214	recursive co-expression activation
301	interpreter stack overflow
302	C stack overflow
303	unable to expand memory region
304	memory region size changed
305	inadequate space in string region
306	inadequate space in block region
307	inadequate space for icode
308	inadequate space for i/o buffers
309	inadequate space for interpreter stack