Version 5.10 of Icon*

Ralph E. Griswold

William H. Mitchell

TR 85-16

August 22, 1985

Corrected September 14, 1985

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 5.10 of Icon

1. Introduction

Reference 1 describes the standard Version 5 of Icon. Since Icon is the byproduct of a research effort that is concerned with the development of novel programming language facilities for processing nonnumeric data, it is inevitable that some extensions to the standard language will develop.

These extensions are incorporated as features of new releases, which are identified by minor version numbers such as 5.8, 5.9, and 5.10. This report describes the extensions that are included in Version 5.10 of Icon.

These extensions are upward-compatible with standard Version 5 Icon. Their inclusion should not interfere with any program that works properly under the standard version.

2. Version 5.10 Changes

The following sections describe all the extensions to the standard features of Version 5 of Icon. For reference, the following features in Version 5.10 are changes from Version 5.9:

- Extensions are no longer optional; all extensions are contained in all implementations of Version 5.10.
- There are new options for sorting tables.
- The function collect() is new.
- There are restrictions on the use of string invocation for operators that were not present in Version 5.9.
- There are some minor changes in the terminology used in error messages.

3. Version 5.10 Features

3.1 The Link Directive

The link directive simplifies the inclusion of separately translated libraries of Icon procedures. If *icont(1)* [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 directive, 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. Syntax appropriate to VMS should be used when running under that system.

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

The environment variable *IPATH* controls the location of files specified in link directives. *IPATH* should have a value of the form *p1:p2:* ... *pn* where each *pi* names a directory. Each directory is searched in turn to locate files named in link directives. The default value of *IPATH* is '.', that is, the current directory.

3.2 Sets

Sets are unordered collections of values and have 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.

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 and

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

writes set(2). Note that the string images of sets are in the same style as for other aggregates, with the size enclosed in parentheses.

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 themselves occur after tables but before records in the collating order.

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** (it is similar to **put(a, x)** in form). Note that

```
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).

Examples

Word Counting:

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

```
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(a)
   local limit, s, i
   limit := a[1] | 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
```

3.3 Sorting Tables

Two new options are available for sorting tables, and are specified by the values 3 and 4 for 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 much less than for the list of lists, especially if there are many elements in the table.

3.4 Integer Sequences

To facilitate the generation of integer sequences that have no limit, the experimental features include the function **seq(i, j)**. This function has the result sequence {i, i+j, i+2j, ... }. Omitted or null values for i and j default to 1. Thus the result sequence for **seq()** is {1, 2, 3, ... }.

3.5 PDCO Invocation Syntax

The experimental features include the procedure invocation syntax that is used for programmer-defined control operations [3]. In this syntax, when braces are used in place of parentheses to enclose an argument list, the arguments are passed as a list of co-expressions. That is,

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([])

3.6 Invocation Via String Name

The experimental features allow a string-valued expression that corresponds to the name of a procedure or operation to 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 operations, 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 also 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.

3.7 Conversion to Procedure

The experimental features include the function proc(x, i), which 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 exponentiation operator, while proc("^", 1) produces the co-expression refresh operator. If x cannot be converted to a procedure, proc(x, i) fails.

3.8 Installation Options

When an Icon system is installed, various configuration options can be specified [4]. The value of the keyword &options is a string that contains the command line arguments that were used to configure Icon.

3.9 Forced Garbage Collection

Storage is allocated automatically during the execution of an Icon program, and garbage collections are performed as necessary to reclaim storage for subsequent reallocation [5]. Garbage collection normally only occurs when it is necessary. Garbage collection can be forced by the built-in function

```
collect()
```

4. Known Bugs in Version 5.10

- 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 are 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.
- File names are truncated to 14 characters by some versions of UNIX. If such a truncation deletes part of the terminating .icn of a file that is input to the translator, mysterious diagnostic messages may occur during linking.
- There is a bug in the 4.1bsd fopen() routine that under certain conditions returns a FILE pointer that is out of range when an attempt is made to open too many files. On systems where this bug is present, it may manifest itself in the form of run-time Error 304 when an attempt is made to open too many files. (On 4.1bsd systems this limit is usually 20 files.)
- If an expression such as

  ```
  x := create expr
  ```

  is used in a loop, and x is not a global variable, unreferenceable co-expression stacks are generated by each successive create operation. These stacks 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
  ```

- Overflow of a co-expression stack, usually due to excessive recursion, is not detected and may cause mysterious program malfunction.
- Program malfunction may occur if display(f, i) is used in a co-expression.
- The garbage collector was designed for machines with small address spaces and may not perform well for computers with 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 Icon an inordinately long time to determine that the object can not be allocated.

Acknowledgements

The design of sets for Icon was done as part of a class project. In addition to the authors of this paper, the following persons participated in the design: John Bolding, Owen Fonorow, Roger Hayes, Tom Hicks, Robert Kohout, Mark Langley, Rob McConeghy, Susan Moore, Maylee Noah, Janalee O'Bagy, Gregg Townsend, and Alan Wendt.

The implementation of sets in Version 5.10 and the new sorting options were done by Rob McConeghy, who also made a number of improvements to other aspects of the implementation.

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. and William H. Mitchell. *ICONT(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona. August 1985.

3. Griswold, Ralph E. and Michael Novak. "Programmer-Defined Control Operations", *The Computer Journal*, Vol. 26, No. 2 (May 1983). pp. 175-183.

4. Griswold, Ralph E. and William H. Mitchell. *Installation and Maintenance Instructions for Version 5.10 of Icon*, Technical Report TR 85-15, Department of Computer Science, The University of Arizona. August 1985.

5. Griswold, Ralph E. and William H. Mitchell. *A Tour Through the C Implementation of Icon; Version 5.10*, Technical Report TR 85-19, Department of Computer Science, The University of Arizona. August 1985.