An Overview of Version 8 the Icon Programming Language*

Ralph E. Griswold

TR 90-6a

January 1, 1990; last revised February 13, 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.

An Overview of Version 8 the Icon Programming Language

1. Introduction

Icon is a high-level programming language with extensive facilities for processing strings and structures. Icon has several novel features, including expressions that may produce sequences of results, goal-directed evaluation that automatically searches for a successful result, and string scanning that allows operations on strings to be formulated at a high conceptual level.

Icon emphasizes high-level string processing and a design philosophy that allows ease of programming and short, concise programs. Storage allocation and garbage collection are automatic in Icon, and there are few restrictions on the sizes of objects. Strings, lists, and other structures are created during program execution and their size does not need to be known when a program is written. Values are converted to expected types automatically; for example, numeral strings read in as input can be used in numerical computations without explicit conversion. Icon has an expression-based syntax with reserved words; in appearance, Icon programs resemble those of Pascal and C.

Although Icon has extensive facilities for processing strings and structures, it also has a full repertoire of computational facilities. It is suitable for a wide variety of applications. Some examples are:

- text analysis, editing, and formatting
- document formating
- artificial intelligence
- expert systems
- rapid prototyping
- symbolic mathematics
- text generation
- data laundry

There are public-domain implementations of Icon for the Amiga, the Atari ST, the Macintosh under MPW, MS-DOS, MVS, OS/2, many UNIX systems, VM/CMS, and VMS. There also is a commercial implementation of an enhanced version of Icon for the Macintosh [1].

The remainder of this report briefly describes the highlights of Icon. For a complete description, see [2, 3].

2. Expression Evaluation

2.1 Conditional Expressions

In Icon there are *conditional expressions* that may *succeed* and produce a result, or may *fail* and not produce any result. An example is the comparison operation

```
i > j
```

which succeeds (and produces the value of j) provided that the value of i is greater than the value of j, but fails otherwise. Similarly,

```
i > j > k
```

succeeds if the value of j is between i and k.

The success or failure of conditional operations is used instead of Boolean values to drive control structures in Icon. An example is

```
if i > j then k := i else k := j
```

which assigns the value of i to k if the value of i is greater than the value of j, but assigns the value of j to k otherwise.

The usefulness of the concepts of success and failure is illustrated by find(s1,s2), which fails if s1 does not occur as a substring of s2. Thus

```
if i := find("or",line) then write(i)
```

writes the position at which "or" occurs in line, if it occurs, but does not write a value if it does not occur.

Many expressions in Icon are conditional. An example is read(), which produces the next line from the input file, but fails when the end of the file is reached. The following expression is typical of programming in Icon and illustrates the integration of conditional expressions and conventional control structures:

```
while line := read() do
   write(line)
```

This expression copies the input file to the output file.

If an argument of a function fails, the function is not called, and the function call fails as well. This ''inheritance'' of failure allows the concise formulation of many programming tasks. Omitting the optional do clause in while-do, the previous expression can be rewritten as

```
while write(read())
```

2.2 Generators

In some situations, an expression may be capable of producing more than one result. Consider

```
sentence := "Store it in the neighboring harbor"
find("or",sentence)
```

Here "or" occurs in sentence at positions 3, 23, and 33. Most programming languages treat this situation by selecting one of the positions, such as the first, as the result of the expression. In Icon, such an expression is a *generator* and is capable of producing all three positions.

The results that a generator produces depend on context. In a situation where only one result is needed, the first is produced, as in

```
i := find("or",sentence)
```

which assigns the value 3 to i.

If the result produced by a generator does not lead to the success of an enclosing expression, however, the generator is *resumed* to produce another value. An example is

```
if (i := find("or",sentence)) > 5 then write(i)
```

Here the first result produced by the generator, 3, is assigned to i, but this value is not greater than 5 and the comparison operation fails. At this point, the generator is resumed and produces the second position, 23, which is greater than 5. The comparison operation then succeeds and the value 23 is written. Because of the inheritance of failure and the fact that comparison operations return the value of their right argument, this expression can be written in the following more compact form:

```
write(5 < find("or",sentence))
```

Goal-directed evaluation is inherent in the expression evaluation mechanism of Icon and can be used in arbitrarily complicated situations. For example,

```
find("or",sentence1) = find("and",sentence2)
```

succeeds if "or" occurs in sentence1 at the same position as and occurs in sentence2.

A generator can be resumed repeatedly to produce all its results by using the every-do control structure. An example is

```
every i := find("or",sentence)
   do write(i)
```

which writes all the positions at which "or" occurs in sentence. For the example above, these are 3, 23, and 33.

Generation is inherited like failure, and this expression can be written more concisely by omitting the optional do clause:

```
every write(find("or",sentence))
```

There are several built-in generators in Icon. One of the most frequently used of these is

```
i to j
```

which generates the integers from i to j. This generator can be combined with every-do to formulate the traditional for-style control structure:

```
every k := i to j do
   f(k)
```

Note that this expression can be written more compactly as

```
every f(i to j)
```

There are a number of other control structures related to generation. One is *alternation*,

$expr_1$ | $expr_2$

which generates the results of $expr_1$ followed by the results of $expr_2$. Thus

```
every write(find("or",sentence1) | find("or",sentence2))
```

writes the positions of "or" in sentence1 followed by the positions of "or" in sentence2. Again, this sentence can be written more compactly by using alternation in the second argument of find():

```
every write(find("or",sentence1 | sentence2))
```

Another use of alternation is illustrated by

```
(i | j | k) = (0 | 1)
```

which succeeds if any of i, j, or k has the value 0 or 1.

Procedures can be used to add generators to Icon's built-in repertoire. For example,

```
procedure findodd(s1,s2)
   every i := find(s1,s2) do
      if i % 2 = 1 then suspend i
end
```

is a procedure that generates the odd-valued positions at which s1 occurs in s2. The suspend control structure returns a value from the procedure, but leaves it in suspension so that it can be resumed for another value. When the loop terminates, control flows off the end of the procedure without producing another value.

3. String Scanning

For complicated operations, the bookkeeping involved in keeping track of positions in strings becomes burdensome and error prone. Icon has a string scanning facility that is manages positions automatically. Attention is focused on a current position in a string as it is examined by a sequence of operations.

The string scanning operation has the form

s ? *expr*

where s is the *subject* string to be examined and *expr* is an expression that performs the examination. A position in the subject, which starts at 1, is the focus of examination.

Matching functions change this position. One matching function, move(i), moves the position by i and produces the substring of the subject between the previous and new positions. If the position cannot be moved by the specified amount (because the subject is not long enough), move(i) fails. A simple example is

```
line ? while write(move(2))
```

which writes successive two-character substrings of line, stopping when there are no more characters.

Another matching function is tab(i), which sets the position in the subject to i and also returns the substring of the subject between the previous and new positions. For example,

```
line ? if tab(10) then write(tab(0))
```

first sets the position in the subject to 10 and then to the end of the subject, writing line[10:0]. Note that no value is written if the subject is not long enough.

String analysis functions such as find() can be used in string scanning. In this context, the string that they operate on is not specified and is taken to be the subject. For example,

```
line ? while write(tab(find("or")))
    do move(2)
```

writes all the substrings of line prior to occurrences of "or". Note that find() produces a position, which is then used by tab to change the position and produce the desired substring. The move(2) skips the "or" that is found.

Another example of the use of string analysis functions in scanning is

```
line ? while tab(upto(&letters)) do
    write(tab(many(&letters)))
```

which writes all the words in line.

As illustrated in the examples above, any expression may occur in the scanning expression.

4. Structures

Icon supports several kinds of structures with different organizations and access methods. Lists are linear structures that can be accessed both by position and by stack and queue functions. Sets are collections of arbitrary values with no implied ordering. Tables provide an associative lookup mechanism.

4.1 Lists

While strings are sequences of characters, lists in Icon are sequences of values of arbitrary types. Lists are created by enclosing the lists of values in brackets. An example is

```
car1 := ["buick","skylark",1978,2450]
```

in which the list car1 has four values, two of which are strings and two of which are integers. Note that the values in a list need not all be of the same type. In fact, any kind of value can occur in a list — even another list, as in

```
inventory := [car1,car2,car3,car4]
```

Lists also can be created by

```
L := list(i,x)
```

which creates a list of i values, each of which has the value x.

The values in a list can be referenced by position much like the characters in a string. Thus

```
car1[4] := 2400
```

changes the last value in car1 to 2400. A reference that is out of the range of the list fails. For example,

```
write(car1[5])
```

fails.

The values in a list L are generated by !L. Thus

```
every write(!L)
```

writes all the values in L.

Lists can be manipulated like stacks and queues. The function push(L,x) adds the value of x to the left end of the list L, automatically increasing the size of L by one. Similarly, pop(L) removes the leftmost value from L, automatically decreasing the size of L by one, and produces the removed value.

4.2 Sets

A sets is a collection of values. An empty set is created by set(). Alternatively, set(L) produces a set with the values in the list L. For example,

```
S := set([1,"abc",[]])
```

assigns to S a set that contains the integer 1, the string "abc", and an empty list.

The set operations of union, intersection, and difference are provided. The function member(S,x) succeeds if x is a member of the set S but fails otherwise. The function insert(S,x) adds x to the set S, while delete(S,x) removes x from S. A value only can occur once in a set, so insert(S,x) has no effect if x is already in S. !S generates the members of S.

A simple example of the use of sets is given by the following segment of code, which lists all the different words that appear in the input file:

```
words := set()
while line := read() do
   line ? while tab(upto(&letters)) do
      insert(words,tab(many(&letters)))
every write(!words)
```

4.3 Tables

Tables are sets of pairs each of which consists of a key and a corresponding value. The key and its corresponding value may be of any type, and the value for any key can be looked up automatically. Thus, tables provide a form of associative access in contrast with the positional access to values in lists.

A table is created by an expression such as

```
symbols := table(0)
```

which assigns to symbols a table with the default value 0. The default value is used for keys that are not assigned another value. Subsequently, symbols can be referenced by any key, such as

```
symbols["there"] := 1
```

which assigns the value 1 to the key "there" in symbols.

Tables grow automatically as new keys are added. For example, the following program segment produces a table containing a count of the words that appear in the input file:

```
words := table(0)
while line := read() do
   line ? while tab(upto(&letters)) do
      words[tab(many(&letters))] +:= 1
```

Here the default value for each word is 0, as given in table(0), and +:= is an augmented assignment operation that increments the values by one. There are augmented assignment operations for all binary operators.

A list can be obtained from a table T by the function sort(T,1). The form of the list depends on the value of i. For example, if i is 3, the list contains alternate keys and their corresponding values in T. For example,

```
wordlist := sort(words,3)
while write(pop(wordlist)," : ",pop(wordlist))
```

writes the words and their counts from words.

5. An Example

The following program, which produces a concordance of the words from an input file, illustrates typical Icon programming techniques. Although not all the features in this program are described in previous sections, the general idea should be clear.

```
global uses, lineno, width

procedure main(args)
   width := 15                          # width of word field
   uses := table()
   lineno := 0
   every tabulate(words())              # tabulate all the citations
   output()                             # print the citations
end

#  Add line number to citations for word
#
procedure tabulate(word)
   /uses[word] := table()
   uses[word][lineno] := 1
   return
end

#  Generate words
#
procedure words()
   while line := read() do {
      lineno +:= 1
      write(right(lineno,6),"   ",line)
      map(line) ? while tab(upto(&letters)) do {
         s := tab(many(&letters))
         if *s < 3 then next            # skip short words
         suspend s
         }
      }
end
```

```
#   Print the results
#
procedure output()
   write()                                     # blank line
   uses := sort(uses,3)                        # sort citations
   while word := get(uses) do {
      line := ""
      numbers := sort(get(uses),3)
      while line ||:= get(numbers) || ", " do
         get(numbers)                          # skip marking value
      write(left(word,width), line[1:–2])
      }
end
```

The program reads a line, writes it out with an identifying line number, and processes every word in the line. Words less than three characters long are considered to be ‘‘noise’’ and are discarded. A table, uses, is keyed by the words. Every key has a corresponding set of line numbers. The first time a word is encountered, a new set is created for it. The line number is inserted in any event. Since a value can be in a set only once, duplicate line numbers are suppressed automatically.

After all the input has been read, the table of words is sorted by key. Each corresponding set of line numbers is sorted and the line numbers are appended to the line to be written.

For example, if the input file is

```
        On the Future!–how it tells
        Of the rapture that impells
       To the swinging and the ringing
        Of the bells, bells, bells–
     Of the bells, bells, bells, bells,
               Bells, bells, bells–
 To the rhyming and the chiming of the bells!
```

the output is

```
1         On the Future!–how it tells
2         Of the rapture that impells
3        To the swinging and the ringing
4         Of the bells, bells, bells–
5      Of the bells, bells, bells, bells,
6                Bells, bells, bells–
7   To the rhyming and the chiming of the bells!

and          3, 7
bells        4, 5, 6, 7
chiming      7
future       1
how          1
impells      2
rapture      2
rhyming      7
ringing      3
swinging     3
tells        1
that         2
the          1, 2, 3, 4, 5, 7
```

Acknowledgement

Icon was designed by the the author in collaboration with Dave Hanson, Tim Korb, Cary Coutant, and Steve Wampler. Many other persons have contributed to its development. The current implementation is based on the work of Cary Coutant and Steve Wampler with recent contributions by Bill Mitchell, Janalee O’Bagy, Gregg Townsend, and Ken Walker.

References

1. R. E. Griswold and M. T. Griswold, *The ProIcon Programming Language for the Apple Macintosh Computers*, The Bright Forest Company, Tucson, AZ, 1989.
2. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
3. R. E. Griswold, *Version 8 of Icon*, The Univ. of Arizona Tech. Rep. 90-1, 1990.