An Overview of the Icon Programming Language*

Ralph E. Griswold

TR 83-3g

May 13, 1983; last revised April 6, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

An Overview of the Icon Programming Language

1. Introduction

Icon is a high-level programming language with extensive facilities for processing strings and lists. 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 resembles SNOBOL4 [1] in its emphasis on high-level string processing and a design philosophy that allows ease of programming and short, concise programs. Like SNOBOL4, 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. Whereas SNOBOL4 has a pattern-matching facility that is separate from the rest of the language, string scanning is integrated with the rest of the language facilities in Icon. Unlike SNOBOL4, Icon has an expression-based syntax with reserved words; in appearance, Icon programs resemble those of several other conventional programming languages.

Examples of the kinds of problems for which Icon is well suited are:

- text analysis, editing, and formatting
- document preparation
- symbolic mathematics
- text generation
- parsing and translation
- data laundry
- graph manipulation

Version 7 of Icon, the most recent version, is implemented in C [2]. There are UNIX* implementations for many computers, including the Amdahl 580, the AT&T 3B series, the HP 9000, the IBM PC/XT/AT, the IBM RT PC, the Pyramid 90x, the Ridge 32, the Sun Workstation, the UNIX PC, and the VAX-11. There also are implementations for VAX/VMS, MS-DOS, the Amiga, the Atari ST, and the Macintosh. Other implementations are in progress.

A brief description of some of the representative features of Icon is given in the following sections. This description is not rigorous and does not include many features of Icon. See [3] for a complete description and [4] for a description of recent changes to the language.

2. Strings

Strings of characters may be arbitrarily long, limited only by the architecture of the computer on which Icon is implemented. A string may be specified literally by enclosing it in double quotation marks, as in

```
greeting := "Hello world"
```

which assigns an 11-character string to `greeting`, and

```
address := ""
```

which assigns the zero-length *empty* string to `address`. The number of characters in a string `s`, its size, is given by `*s`. For example, `*greeting` is 11 and `*address` is 0.

---

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

Icon uses the ASCII character set, extended to 256 characters. There are escape conventions, similar to those of C, for representing characters that cannot be keyboarded.

Strings also can be read in and written out, as in

```
line := read()
```

and

```
write(line)
```

Strings can be constructed by concatenation, as in

```
element := "(" || read() || ")"
```

If the concatenation of a number of strings is to be written out, the write function can be used with several arguments to avoid actual concatenation:

```
write("(",read(),")")
```

Substrings can be formed by subscripting strings with range specifications that indicate, by position, the desired range of characters. For example,

```
middle := line[10:20]
```

assigns to middle the string of characters of line between positions 10 and 20. Similarly,

```
write(line[2])
```

writes the second character of line. The value 0 is used to refer to the position after the last character of a string. Thus

```
write(line[2:0])
```

writes the substring of line from the second character to the end, thus omitting the first character.

An assignment can be made to the substring of string-valued variable to change its value. For example,

```
line[2] := "..."
```

replaces the second character of line by three dots. Note that the size of line changes automatically.

There are many functions for analyzing strings. An example is

```
find(s1, s2)
```

which produces the position in s2 at which s1 occurs as a substring. For example, if the value of greeting is as given earlier,

```
find("or", greeting)
```

produces the value 8. See Section 4.2 for the handling of situations in which s1 does not occur in s2, or in which it occurs at several different positions.

3. Character Sets

While strings are sequences of characters, *csets* are sets of characters in which membership rather than order is significant. Csets are represented literally using single enclosing quotation marks, as in

```
vowels := 'aeiouAEIOU'
```

Two useful built-in csets are &lcase and &ucase, which consist of the lowercase and uppercase letters, respectively. Set operations are provided for csets. For example,

```
letters := &lcase ++ &ucase
```

forms the cset union of the lowercase and uppercase letters and assigns the resulting cset to letters, while

```
consonants := letters –– 'aeiouAEIOU'
```

forms the cset difference of the letters and the vowels and assigns the resulting cset to consonants.

Csets are useful in situations in which any one of a number of characters is significant. An example is the string analysis function

```
upto(c, s)
```

which produces the position s at which any character in c occurs. For example,

```
upto(vowels, greeting)
```

produces 2. Another string analysis function that uses csets is

```
many(c, s)
```

which produces the position in s after an initial substring consisting only of characters that occur in s. An example of the use of many is in locating words. Suppose, for example, that a word is defined to consist of a string of letters. The expression

```
write(line[1:many(letters, line)])
```

writes a word at the beginning of line. Note the use of the position returned by a string analysis function to specify the end of a substring.

4. Expression Evaluation

4.1 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.

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())
```

4.2 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.

5. String Scanning

The string analysis and synthesis operations described in Sections 2 and 3 work best for relatively simple operations on strings. For complicated operations, the bookkeeping involved in keeping track of positions in strings becomes burdensome and error prone. In such cases, Icon has a string scanning facility that is analogous in many respects to pattern matching in SNOBOL4. In string scanning, positions are managed automatically and 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. Unlike SNOBOL4, in which the operations that are allowed in pattern matching are limited and idiosyncratic, string scanning is completely integrated with the rest of the operation repertoire of Icon.

6. 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.

6.1 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

```
a := 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 a are generated by !a. Thus

```
every write(!a)
```

writes all the values in a.

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

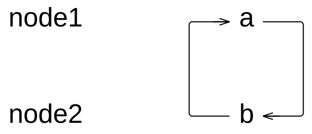
A list value in Icon is a pointer (reference) to a structure. Assignment of a structure in Icon does not copy the structure itself but only the pointer to it. Thus the result of

```
demo := car1
```

causes demo and car1 to reference the same list. Graphs with loops can be constructed in this way. For example,

```
node1 := ["a"]
node2 := [node1, "b"]
push(node1, node2)
```

constructs a structure that can be pictured as follows:



6.2 Sets

Sets are collections of values. A set is obtained from a list by `set(a)`, where `a` contains the members of the set. 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`.

The operation `*s` produces the number of members in `s` and `!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)
```

6.3 Tables

Icon has a table data type similar to that of SNOBOL4. Tables essentially are sets of pairs of values, an *entry value* and a corresponding *assigned value*. The entry and assigned values may be of any type, and the assigned value for any entry value 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(x)
```

which assigns to `symbols` a table with the default assigned value `x`. Subsequently, `symbols` can be referenced by any entry value, such as

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

which assigns the value 1 to the `there`th entry in symbols.

Tables grow automatically as new entry values 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 assigned value for each word is 0, as given in `table(0)`, and `+:=` is an augmented assignment operation that increments the assigned values by one. There are augmented assignment operations for all binary operators.

A list can be obtained from a table 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 entry and assigned values of `t`. For example,

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

writes the words and their counts from words.

7. Procedures

An Icon program consists of a sequence of procedure declarations. An example of a procedure declaration is

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

where the name of the procedure is max and its formal parameters are i and j. The **return** expressions return the value of i or j, whichever is larger.

Procedures are called like built-in functions. Thus

```
k := max(*s1, *s2)
```

assigns to k the size of the longer of the strings s1 and s2.

A procedure also may suspend instead of returning. In this case, a result is produced as in the case of a return, but the procedure can be resumed to produce other results. An example is the following procedure that generates the words in the input file.

```
procedure genword()
   local line, letters, words
   letters := &lcase ++ &ucase
   while line := read() do
      line ? while tab(upto(letters)) do {
         word := tab(many(letters))
         suspend word
         }
end
```

The braces enclose a compound expression.

Such a generator is used in the same way that a built-in generator is used. For example

```
every word := genword() do
   if find("or", word) then write(word)
```

writes only those words that contain the substring or.

8. An Example

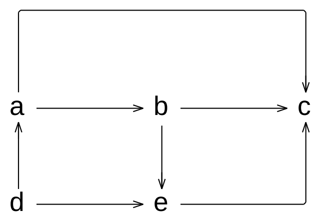
The following program sorts graphs topologically.

```
procedure main()
   local sorted, nodes, arcs, roots
   while nodes := read() do {                          # get next node list
      arcs := read()                                   # get arc list
      sorted := ""                                     # sorted nodes
                                                       # get nodes without predecessors
      while *(roots := nodes -- snodes(arcs)) > 0 do {
         sorted ||:= roots                             # add to sorted nodes
         nodes --:= roots                              # delete these nodes
         arcs := delarcs(arcs, roots)                  # delete their arcs
         }
      if *arcs = 0 then write(sorted)                  # successfully sorted
      else write("graph has cycle")                    # cycle if node remains
   }
end

procedure snodes(arcs)
   local nodes
   nodes := ""
   arcs ? while move(1) do {                           # predecessor
      move(2)                                          # skip "->"
      nodes ||:= move(1)                               # successor
      move(1)                                          # skip ";"
      }
   return nodes
end

procedure delarcs(arcs, roots)
   local newarcs, node
   newarcs := ""
   arcs ? while node := move(1) do {                   # get predecessor node
      if many(roots, node) then move(4)                # delete arc from root node
      else newarcs ||:= node || move(4)                # else keep arc
      }
   return newarcs
end
```

Graph nodes are represented by single characters with a list of the nodes on one input line followed by a list of arcs. For example, the graph



is given as

```
abcde
aXXXX->b;aXXXX->c;bXXXX->c;bXXXX->e;dXXXX->a;dXXXX->e;eXXXX->c;
```

for which the output is

```
dabec
```

The nodes are represented by csets and automatic type conversion is used to convert strings to csets and vice versa. Note the use of augmented assignment operations for concatenation and in the computation of cset differences.

Acknowledgement

Icon was designed by the the author in collaboration with Dave Hanson, Tim Korb, Cary Coutant, and Steve Wampler. The current implementation is largely the work of Cary Coutant and Steve Wampler with recent contributions by Bill Mitchell and Janalee O'Bagy. Dave Hanson and Bill Mitchell made several helpful suggestions on the presentation of material in this paper.

References

1. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language*, second edition. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1971.
2. Griswold, Ralph E. and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, New Jersey. 1986.
3. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.
4. Griswold, Ralph E., Gregg M. Townsend, and Kenneth Walker. *Version 7 of Icon*, Technical Report TR 88-5, Department of Computer Science, The University of Arizona. 1988.