The Icon Programming Language An Overview*

Ralph E. Griswold, David R. Hanson, and John T. Korb

TR 78-3d

December 1978
Revised March 1979

Department of Computer Science
The University of Arizona

*This work was supported in part by the National Science Foundation under Grant MCS75-01307.

The Icon Programming Language
An Overview

1. Introduction

Icon is a programming language designed for nonnumerical applications with an emphasis on string processing. Icon inherits the philosophical bases of SNOBOL4 [1] and SL5 [2]: high-level facilities, novel features, ease and convenience of use, and runtime flexibility.

Icon emphasizes expressive power in control structures. It resembles SL5 rather than SNOBOL4 in this respect, although new mechanisms allow a smaller repertoire of basic constructs than that of SL5. Like SL5, Icon expressions return a result consisting of a value and a signal. Values are used in the normal computational manner, while signals drive control structures.

An important component of Icon is goal-directed evaluation of expressions, called generators. Generators are capable of producing alternative values as demanded by circumstances. The goal-directed control structures and generators facilitate programming of search algorithms and they eliminate the need for patterns and scanning environments used for string analysis in SNOBOL4 and SL5. In Icon, string scanning operations can be freely mixed with other operations, integrating string processing as a natural part of the language, as opposed to a separate facility as in SNOBOL4 and SL5 [3].

Like SNOBOL4 and SL5, Icon has numerous data types. In addition to the conventional numerical and string types, there are character sets, lists, stacks, tables, and records.

The remainder of this paper provides a brief overview of the most important aspects of Icon. See Reference 4 for a complete description. The reader is assumed to be familiar with SNOBOL4 (in general) and SL5 (for particulars).

2. Data Types and Structures

The built-in Icon types are:

integer
real
string
cset
file
procedure
list
table
stack
null

The cset (character set) type is an addition to the SNOBOL4-SL5 repertoire and is used in situations where strings are used as sets of characters, as in BREAK(S) in SNOBOL4. The table type is similar to that in SNOBOL4, although some additional operations on tables are supported in Icon. Stacks are also provided as a built-in feature in Icon. Character sets, lists, tables, stacks, and record types are described in more detail in subsequent sections. The null type provides a canonical representation for null values of various types, such as the null string.

As in SNOBOL4 and SL5, type conversions are performed implicitly where required by context. For example,

```
write(2.7)
```

converts the real number 2.7 to a string for the purposes of output.

In addition, there are explicit type-conversion functions. For example

```
average := string(2.7)
```

assigns to average a string corresponding to the real number 2.7.

3. Control Structures

Control structures in Icon are similar to those in SL5. Reserved words and operators are used for the syntax.

An example of increased expressive power is given by the suffix reserved word **fails**, which has the effect of inverting the signal of the expression that it follows. Thus

```
if e1 fails then e2
```

subsumes the SL5 expression

```
unless e1 do e2
```

Generators most noticeably distinguish Icon from SNOBOL4 and SL5. In Icon, the mode of evaluation is oriented toward seeking successful results in the presence of alternative values.

Alternatives may be specified explicitly by the alternation operator, e1 | e2. Unlike SNOBOL4 and SL5, where this operator constructs a pattern or scanning environment, in Icon this operator more closely resembles logical disjunction "either e1 or e2". That is, either e1 or e2 are possible values of this expression. For example

```
(x | y) > 3
```

succeeds if either x or y is greater than 3. This is equivalent to

```
(x > 3) | (y > 3)
```

In both cases, alternatives are generated as they occur from left to right. Alternatives may be arbitrarily complex, as in

```
(x | y | z) > (a | b | c | d)
```

which succeeds if the value of any of the identifiers on the left side of the comparison operation is greater than the value of any of the identifiers on the right.

The mutual success of two expressions can be specified by the conjunction operator, e1 & e2. This operation succeeds only if both e1 and e2 succeed. For example

```
((x + y) > 5) & ((x + z) > y)
```

succeeds only if both of the specified conditions hold. If e1 and e2 have alternatives, the evaluation of the conjunction assures that all alternatives are evaluated in the attempt to mutually satisfy e1 and e2. For example

```
(x := (3 | 4)) & (x + 2 > 5)
```

results in x being assigned the value 4. Since evaluation of alternatives is from left to right, x is first assigned the value 3, but when the second operand of the conjunction fails, the first operand is evaluated again, causing 4 to be assigned to x.

Goal-directed evaluation does not reverse the effects of previous operations. For example,

```
(x := (3 | 4)) & (x + 2 > 6)
```

fails, but leaves 4 assigned to x.

The **every** construct produces all alternatives of an expression and evaluates a **do** clause for each alternative. An example is

```
every i := (1 | 3 | 7) do f(i)
```

which evaluates f(1), f(3), and f(7).

In Icon, the expression

```
e1 to e2 by e3
```

is a generator so that

```
every i := e1 to e2 by e3 do e4
```

replaces the traditional **for** statement. While the **to** generator is typically used in **every** constructs, it may be used anywhere that a generator is meaningful. Similarly, any generator can be used in the **every** construct, allowing the composition of complex control structures from simpler ones.

Another generator is !x, which generates the elements of x in sequence. If x is a string, successive characters are generated. If x is an list, its elements are generated in numerical sequence. The operator applies to other types as well, providing a uniform mechanism for processing all elements of a structure. For example

```
every write(!x)
```

prints the value of every element in the structure x.

4. Keywords

Icon keywords are similar to those of SNOBOL4 and SL5, and serve as a communication interface between the running program and the Icon system. Typical keywords are:

&ascii	string of all ASCII characters
&clock	time of day
&cset	cset of all available characters
&date	date
&lcase	lower-case letters
&level	level of procedure call
&time	elapsed run time
&trace	limit on tracing
&ucase	upper-case letters

5. String Operations

The positions of characters in a string are numbered from the left starting at 1. The numbering identifies positions between characters. For example, the positions in the string CAPITAL are

```
 C A P I T A L
 ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
 1 2 3 4 5 6 7 8
```

Note that the position after the last character may be specified.

Positions may also be specified with respect to the right end of a string, using nonpositive numbers starting at 0 and continuing with negative values toward the left:

```
 C A P I T A L
 ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
-7 -6 -5 -4 -3 -2 -1 0
```

For this string, positions 8 and 0 are equivalent, positions -1 and 7 are equivalent, and so on.

5.1 Basic String Operations

The basic string operations of Icon are similar to those of SL5, although there are a few differences. For example, substr(s,i,l) returns the substring of s starting at position i and of length l, while section(s,i,j) returns the substring of s between positions i and j, inclusive. The expression s[i] references the character to the right of position i in the string s.

There is a repertoire of other functions that operate on strings. For example, the value of size(s) is the number of characters in s and the value of repl(s,n) is a string consisting of n replications of s.

Some string operations act as generators in Icon. An example is upto(c,s), where the value returned is the position in s at the first occurrence of a character in c. Since there may be more than one occurrence of a character in c in this string, there is generally more than one possible value of the expression upto(c,s). These values are generated (in increasing sequence) as needed. For example,

```
every i := upto(c,s) do write(i)
```

prints the location of every character in c that appears in s. Another string operation that is a generator is find(s1,s2), which returns the locations at which s1 appears as a substring of s2.

5.2 String Scanning

With generators and a mode of evaluation that seeks successful result, much of the motivation for patterns in SNOBOL4 (and scanning environments in SL5) is eliminated. Removing patterns and scanning environments has a number of beneficial aspects:

(1) The sharp distinction between pattern matching and other language operations is removed, permitting string analysis to be intermixed with other operations.

(2) Duplication of control structures, such as the SL5 **or** and the scanning environment for | , is avoided. Furthermore, all control structures can be used during string scanning.

(3) Awkward binding times are avoided, especially where the values of parameters are not known when a pattern or scanning environment is constructed, requiring use of unevaluated expressions or deferred evaluation.

The remaining advantage of pattern matching lies primarily in the suppression of clerical detail resulting from a subject to which operations apply without explicit mention and the implicit cursor movement that obviates bookkeeping.

In Icon, these advantages are retained by allowing a subject and a position for scanning to be established in the same implicit fashion as it occurs in SNOBOL4 and SL5.

The control structure

```
scan s using e
```

establishes s as the current subject of scanning and then evaluates the expression e. The subject is automatically assigned to the keyword &subject and the value 1 is assigned to &pos, corresponding to the beginning of &subject. While the subject and position can be manipulated directly through these two keywords, there are a number of scanning operations that operate implicitly. A typical scanning operation is move(n). Unlike its counterparts in SNOBOL4 and SL5 (which contruct a pattern and scanning environment, respectively), in Icon move(n) simply adds n to &pos and returns as value the substring of &subject between the old and new values of &pos. For example,

```
scan "fleurons" using repeat write(move(2))
```

prints the pairs of characters

```
fl
eu
ro
ns
```

The **repeat** loop is terminated when move can no longer advance the cursor by 2.

Another scanning function is tab(n), which sets &pos to n and returns as value the substring between the old and new values of &pos.

All the basic string analysis operations that specify a string to be examined, e.g. upto(c,s) can be used in string scanning by omitting the string specification. If this is done, the value of s is assumed to be &subject starting at &pos. For example

```
t := tab(upto("aeiou"))
```

assigns to t the portion of &subject from the current value of &pos up to the first vowel. Note that upto("aeiou") simply returns an integer; it does not change the position, which is done by tab. Similarly,

```
scan s using {
  t := ""
  while t := t || tab(upto(" ")) do tab(many(" "))
  }
```

assigns to t the result of deleting all blanks from s.

Note that any language operation can be used in string scanning. In the example above, this includes assignment, **while**, concatenation, in addition to the string operations. The usefulness of this mechanism is particularly evident when the intermediate values of scanning are to be processed. An example is

```
scan s using repeat write(process(move(2)))
```

where process(s) is some defined procedure. Compare this to the methods that are required in SNOBOL4 and SL5.

6. Procedures

An Icon program is a sequence of procedure and record declarations. Procedures are declared in the form

```
procedure
  <header>
  <declarations>
  <body>
end
```

An example is

```
procedure clear(x,i) local j
  j := 0
  repeat x[j+] := i
  return
end
```

which declares clear to be a procedure. The identifiers x and i are formal parameters and j is a local identifier within the procedure.

A procedure may suspend execution rather than returning. The difference between **suspend** and **return** is that a suspended procedure can be reactivated at the point of suspension. Thus procedures may serve as programmer-defined generators. An example is

```
procedure suffix(s,n) local i
  every i := n to size(s) do suspend section(s,i)
  fail
end
```

This procedure generates successively shorter suffixes of the string s, starting at n. An example of its use is

```
every t := suffix(s,1) do write(t)
```

which prints all the suffixes of s.

There are two kinds of identifier declarations: scope and retention. The form of a declaration is

```
<scope> <retention> <identifier>
```

The various declarations are optional and have defaults, but at least a scope or retention declaration must be present.

Scope declarations limit the accessibility of identifiers. There are two scope declarations: **local** and **global**. A local identifier is accessible only within the invocation of the procedure in which it is declared. Global identifiers are accessible to all procedure invocations. An example of a scope declaration is

```
local s
```

which declares s to be a local identifier.

Undeclared identifiers ordinarily default to **local**, although the default may be changed to **global** or **error**. If the default is **error**, undeclared identifiers are diagnosed as programming errors.

The retention declarations are **dynamic** and **static**. The **dynamic** declaration, which is the default, causes storage to be allocated for local identifiers on each invocation of the procedure in which they appear. The **static** declaration is similar to Algol **own**, and causes storage to be allocated permanently so that the value of the corresponding identifier is retained from one invocation of the procedure to the next. Formal parameters are **local** and **dynamic**.

7. Character Sets

Character sets are used in functions such as upto(c,s) where individual characters, but not their order, are important. If a string is given in a context where a cset is expected, type conversion occurs automatically.

The value of &cset is a cset containing all available characters, of which there are 256. When a cset is converted to a string, either implicitly or explicitly, the result is in alphabetical order (collating sequence). For example, the value of string(&cset) is a string of 256 characters. Icon is essentially ASCII based and the first 128 characters in the collating sequence correspond to the ASCII character set. (For computers with character sets other than ASCII, conversion between the external and internal character sets is performed at the input/output interface.)

There are four operations on character sets:

~c	complement with respect to &cset
c1 ++ c2	union
c1 ** c2	intersection
c1 -- c2	difference

8. Lists

Icon lists are created by a reserved word construction that has the form

list <prototype> <initial clause>

This construction creates a list described by the prototype, which is similar to that of SNOBOL4. initial clause is optional and may be used to specify the initial value of all list elements. For example,

```
x := list size(s) initial 0
```

creates a list with as many elements are there are characters in s. Each element has a value that is initially zero.

One-origined lists can be specified literally by giving the list element values in order with surrounding angular brackets. For example

```
x := <1.0,4.0,6.0>
```

assigns to x a list containing three real numbers.

Lists are referenced as in SNOBOL4. An example is

```
x[2] := x[3] * 3.14159
```

Out-of-range list references ordinarily fail. A list can be made expandable by the function open(x). In expandable mode, assignment to one position past the current length of the list causes the list to be extended automatically. A list of size zero is automatically open. For example

```
x := list 0
i := 0
repeat x[i+] := read()
close(x)
```

fills x with values from the input file, automatically extending it. When the input is complete, the list is closed so that further out-of-range references fail.

9. Stacks

Stacks, with the conventional last-in, first-out access method, are created in a fashion similar to lists. The form is

```
stack <size>
```

The size limits the depth of the stack. A zero (or omitted) size produces a stack of unlimited depth. The usual access functions are available: push(k,x), pop(k), and top(k).

Like lists, stacks are heterogeneous.

10. Tables

Icon tables are similar to those in SNOBOL4, except that they are created in a manner similar to lists. The form is

```
table <size> <initial clause>
```

The size limits the number of elements in the table. A zero (or omitted) size produces a table of unlimited size.

```
wordcount := table 0
```

assigns to wordcount a table of unlimited size.

Tables are ordinarily open for the addition of new references. A table may be closed by close(t). A reference to a nonexistent entry in a closed table fails. A table may be opened for new references by open(t).

11. Records

Record types are declared in the form

```
record <name>
  <field1>,
  <field2>,
      .
      .
      .
  <fieldn>
end
```

The name specifies a new record type and field1 through fieldn are the names of fields defined on the type. For example,

```
record complex r, i end
```

defines a new type of record, **complex**, with two fields, r and i.

The form for the creation of instances of records is

```
<type> <value list>
```

For example

```
z := complex 2.0, 3.5
```

creates a record of type **complex** with initial values 2.0 and 3.5 for the r and i fields, respectively.

References to fields are made using the infix dot operator. For example,

```
z.r := z.r + 3.0
```

adds 3.0 to the r field of z.

Records can also be accessed like lists. For example

```
z[1] := z[1] + 3.0
```

is equivalent to the expression above.

12. Input and Output

The function open(s) opens a file corresponding to the name s and returns this file as value. Similarly, a file is closed by close(f). A file that has been closed and reopened is automatically rewound. Note that open(x) and close(x) are polymorphous, and can be applied to files, lists, or tables.

The function write(f,s1,s2, ..., sn) writes the strings s1, s2, ..., sn in sequence onto file f. If f is omitted, the standard output file is assumed. That is, write(s1,s2, ..., sn) writes s1, s2, ..., sn onto the standard output file. This function writes a complete line, appending a newline sequence automatically.

The function read(f) reads a line from file f. If f is omitted, the standard input file is assumed.

13. Programming Example

A simple example of the use of Icon is illustrated by the following procedures which print the intersections at which two words have characters in common [5].

```
procedure cross(word1,word2) local j,k
  every j := upto(word2,word1) do
    every k := upto(word1[j],word2) do
      xprint(word1,word2,j,k)
    return
end
```

```
procedure xprint(word1,,word2,j,k) local pad
  pad := repl(" ",j-1)
  every write(pad,word2[ 1 to k-1 ])
  write(word1)
  every write(pad,word2[ k+1 to size(word2) ])
  write()
  return
end
```

The procedure cross first assigns to j a position in word1 at which a character of word2 is found (the conversion of string to cset for the first argument of upto is done automatically). For this value of j, a position in word2 where the jth character of word1 occurs is then assigned to k. The procedure xprint merely prints the intersection, with word1 being displayed horizontally and word2 being displayed vertically. The uses of **every** insure that all intersections are located.

As an example of the use of these procedures,

```
cross("fish","school")
```

produces the output

```
fish
  c
  h
  o
  o
  l


   s
   c
fish
   o
   o
   l
```

14. Status of Icon

Icon is implemented in Ratfor [6]. Implementations for the DEC-10 and CYBER 175 are presently running and in use.

The system consists of a translator (written in Ratfor) that produces Fortran code. This code, compiled and linked with a library of runtime routines written in Ratfor, produces an executable program.

Only a few of the runtime routines depend on specific computer architecture and system considerations. The system is designed to be portable and a number of other implementations are planned.

15. Acknowledgement

A number of persons have contributed to the Icon language. Cary Coutant, Robert Snyder, and Bruce Trumbo have made especially helpful suggestions. We particularly would like to thank Walt Hansen and Steve Wampler for their help in all phases of the development of the language, particularly in the implementation.

References

1. Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language*, 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey. 1971.

2. Ralph E. Griswold, David R. Hanson, and John T. Korb. "An Overview of SL5", *SIGPLAN Notices*. Vol. 12, No. 5 (April 1977), pp. 40-50.

3. Ralph E. Griswold *An Alternative to the Concept of "Pattern" in String Processing*. Technical Report TR 78-4, Department of Computer Science, The University of Arizona. April 10, 1978.

4. Ralph E. Griswold and David R. Hanson. *Reference Manual for the Icon Programming Language*. Technical Report TR 79-1, Department of Computer Science, The University of Arizona. January 9, 1979.

5. Wetherell, Charles. *Etudes for Programmers*. Prentice-Hall, Englewood Cliffs, New Jersey. 1978. pp. 30-31.

6. Kernighan, Brian W. and P. J. Plauger. *Software Tools*. Addison-Wesley, Reading, Massachusetts. 1976.