Icon Implementation Notes*

David R. Hanson
Walter J. Hansen

TR 79-12a

ABSTRACT

Icon is a new general-purpose programming language intended for nonnumeric applications, especially those involving string and structure processing. This report describes some aspects of the Ratfor implementation of Icon, Version 2.0. Included are a brief overview of the implementation, an explanation of the translator organization and generated code, a description of the runtime environment, and a summary of programming conventions and peculiarities.

July 1979, revised February 1980

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

Icon Implementation Notes

1. Introduction

Icon is a new general-purpose programming language intended for nonnumeric applications, especially those involving string and structure processing. Much of the philosophical basis of Icon comes from its predecessors, SNOBOL4 [1] and SL5 [2]. This report describes some aspects of the implementation of Version 2.0 of Icon. Further details concerning the language and its use may be found in Refs. [3-5]; additional implementation details are given in Refs. [5,6].

The Icon system consists of two major parts: a translator and a runtime system. The translator compiles an Icon program into a program in the target language. The runtime system contains all of the routines referenced by this program. Currently, Fortran is the target language. Thus, running an Icon program requires translating it into Fortran, compiling the Fortran program, and loading it with the previously compiled routines from the runtime system.

2. The Translator

The translator transforms Icon source text into Fortran code. This code consists mainly of calls to subroutines that implement the semantics. Each Icon procedure is translated into a corresponding Fortran subroutine. There is also a Fortran subroutine, generated by the translator, that controls transfer of control. The format of the generated subroutines is described in Appendix A.

The translator consists of three parts: the lexical analyzer, the parser, and the code generator.

2.1 The Lexical Analyzer

The lexical analyzer makes two passes. The first pass builds the symbol table, procedure and record blocks, and resolves undeclared identifiers. In addition, it processes include statements and generates a listing if requested. The results of the first pass are data structure tables and a file containing the source text broken down into tokens (no declaration information is included). The format of each line of the file is

```
token  subtype
```

where "token" is a number representing a token and "subtype" is a number representing additional information about the token. For example, if token represents a literal (INT, FLOAT, or STRING), subtype is the index of the literal in the appropriate literal table. If token represents an operator class (INFIX, PREFIX, SUFFIX), subtype describes the specific operator. If token signifies an identifier (ALPHA), subtype is the index of the identifier in the identifier table. If token represents a keyword (LEXKEY), subtype identifies the specific keyword. Finally, subtype is the same as token for reserved words and keywords. Locations of newlines in the source program are also noted in the token file by including a token representing newline (NEWLINE) with the line number as the value of subtype.

The second pass reads from the token file a token at a time as requested by the parser.

2.2 The Parser

The parser is a recursive descent parser, derived from a context-free grammar for the language. In general, this grammar contains left-recursive productions. These are transformed using the technique described in Ref. [7]. As an example of this technique, the production (note the left recursion)

```
        MULOP    -->    MULOP '*' EXPOP
```

becomes

```
        MULOP    -->    EXPOP MULOPP
        MULOPP   -->    '*' EXPOP MULOPP
                      |  eps
```

Appendix B contains a transformed grammar for Icon.

Recursive descent is usually implemented by writing a recursive procedure for each nonterminal in the grammar [8]. Since Icon is implemented in Ratfor, implementation of recursive procedures was not possible. Instead, a methodology that explicitly simulates recursion is used. As a result, the entire parser is contained in one subroutine. Recursive procedures are simulated using labeled code segments, explicit push and pop operations, and a computed goto for dispatching to return points. The general form of the parser subroutine is as follows.

```
repeat {
   call pop(case)
   goto (...
      labels for all nonterminal routines
      and all possible return points
      ...), case

nonterminal1
      ... code ...
      next

nonterminal2
      ... code ...
      call push(return21)
      goto nonterminalx
return21
      ... code ...
      next
       ...

nonterminaln
      ... code ...
      next
   }
```

The nonterminal code segments are written in much the same way as would recursive procedures in the usual implementation of recursive descent. As the code segments above indicate, the main difference is the need to stack the return label prior to "calling" a code segment for another nonterminal. For example, the code segment for multiplication operators derived from the syntax fragment given above is as follows.

```
MULOP
   call push(MULOPP)
   goto EXPOP          # "call"
MULOPP
   while (token == OMUL) {
      call push(MULOPP1)
      goto EXPOP
MULOPP1
      continue
      }
   next
```

The parser operates on a single Icon procedure and returns a parse tree for that procedure. The parse tree is a directed graph of connected nodes generated during parsing. Each node may have from one to eight fields depending on the semantics of the object represented by that node. The information placed in these fields is constrained by type: labels, pointers, and data. Data may not be placed in pointer fields nor pointers placed in data fields. The value of the variable ptree is the pointer to the current parse tree. When invoking other nonterminal functions the value of ptree must be saved on the stack along with the return label. Unlike the return label, which is popped automatically, the saved value ptree must be popped explicitly. A node is assigned to ptree upon the completion of the processing for each nonterminal. For example, for the multiplicative operators the "continue" in the above code is replaced by

```
        ptree = node5(CINFIX, t, tn, e, ptree)
```

where e is a pointer to the parse tree for the left side of the operator and ptree (inside the function call) is a pointer to the right side.

In addition to ptree, there are several other important variables global to the parser.

The value of the variable fail indicates whether or not an expression can fail. As described in Section 3.3, expressions that can fail require additional code to drive them. This, in turn, requires that an additional node be produced during parsing to indicate the need for the driving code. Thus, many of the nonterminal code segments in the parser save and restore fail if necessary.

The value of the variable var indicates whether not an expression is a variable. This value is used in those code segments that deal with operators, such as assignment, that change the value of program variables.

The value of the variable type indicates the data type of the current expression. Type is used to determine which runtime type conversions, if any, are required.

The values of the variables brklab and nxtlab contain the labels used in break and next expressions. A zero value indicates an improper context for break or next.

2.3 The Code Generator

The code generator produces the Fortran code for a single Icon procedure. It takes as input the parse tree and generates code while traversing the tree in preorder. Since tree traversal is a recursive process, the recursion is simulated using a technique similar to that used in the parser. The targets of the recursive calls in the parser are determined by the grammar and are constant. The targets in the code generator, however, are determined by the configuration of the parse tree. This is handling by placing the label of the next code segment on the stack in addition to the return point. The general form of the code generator is as follows.

```
call push(CDONE)              ≠ set end marker
call push(heap(ptree+NTYPE))  ≠ type of first node
while (ptree > 0) {
   call pop(case)             ≠ current node or label
   goto (...
      labels of parse tree nodes
      and of return points
      ...), case

Cnode
   el = heap(ptree+NPTR1)     ≠ leftmost subtree
   call push(ptree)           ≠ save current parse tree
   call push(Cnodel)          ≠ save return point
   call push(heap(el+NTYPE))  ≠ type of first node
   ptree = el                 ≠ reset parse tree
   next
Cnodel
   call pop(ptree)            ≠ restore parse tree
   call printx("C output code$n$0", 0, 0)
   next                       ≠ "return"
    ...

CDONE                         ≠ end marker
   break
   }
```

Appendix C gives the form of the generated code for each language construct.

3. The Runtime System

The bulk of the Fortran program generated by the translator consists of calls into the runtime system. The runtime system is a set of subroutines that implements the built-in operations in the language. There are also many utility subroutines, including the storage management subsystem, for example.

Operation of the runtime system revolves around a system stack. Functions in the runtime system get their arguments from the stack and leave results there. Activation records for procedures are also placed on the stack. In addition, the storage management system uses the stack as a starting point for locating accessible data objects during reclamation.

Generators are implemented using the common two-stack model for implementing backtracking [9]. The second stack, called the control stack, is used to hold information associated with dormant generators. The system stack could be used to hold this information without the elaborate threading [10] required in more general cases. This simplification, which is used in Ucon (a more recent implementation of Icon for the PDP-11), is possible because the semantics of Icon generators imply that the two stacks operate in parallel.

The following sections describe the representation of data, the organization of storage at runtime, and the operation of some of the subroutines in the runtime system.

3.1 Data Representation and Storage Organization

Storage is represented by the Fortran array mem. Mem is divided into five regions roughly corresponding to the division of data types into classes. These regions are depicted in Figure 1.

```
+----------------+
|                |<----- strbas
|     string     |
|     region     |
|                |
|                |
|                |<----- strfrp
|                |
|                |
+----------------+
|                |<----- sqlbas
|     string     |
|    qualifier   |
|     region     |
|                |
+----------------+
|                |<----- intbas
|    integer     |
|     region     |
|                |
|                |
+----------------+
|                |<----- hepbas
|      heap      |
|     region     |
|                |
|                |
|                |<----- hepfrp
|                |
|                |
|                |<----- sp
|     stack      |
|     region     |
|                |
|                |
|                |<----- stkbas
+----------------+
|                |
|                |
|                |<----- theend
+----------------+
```

Figure 1. Storage Regions.

All source-language values have a uniform representation: an index into mem (indices into mem are often referred to as pointers). The type of a value is determined by the region to which it refers, e.g. if the value is within the bounds specified by intbas and hepbas, it is an integer.

The structure, allocation algorithm, and reclamation technique of each region is determined by the kind of data stored in that region.

Integers are represented by pointers into the integer region. Each cell in the region contains one integer. The lower portion of the region houses a range of permanently allocated integers; the typical range is -1 to 100. A linked list of free cells in the integer region is maintained; allocation consists of simply returning the first free integer on this list.

Strings are represented by pointers into the string qualifier region. A string qualifier is a two-cell block containing the length of the string and the character offset from the beginning of mem to the first character in the string. The actual string is stored in the string region [11,12]. A qualifier for the null (zero-length) string is permanently allocated as the first qualifier in the qualifier region. Allocation of qualifiers is similar to allocation of integers; a linked list of free qualifiers is maintained. For the string region, a free space pointer is maintained (strfrp, see Figure 1). Allocation is done by simply incrementing strfrp by the amount of the request.

All aggregates (e.g. lists, tables, records) are stored in the heap. Storage in the heap is allocated in self-identifying blocks. All pointers into the heap point to the head of a block. There are four block layouts, depending on whether the block contains pointers (called "floating addresses") and is varying or fixed size. The four variations are shown in Figure 2.

```
+----------------+        +----------------+
|   block code   |        |   block code   |
+----------------+        +----------------+
| back reference |        |      size      |
+----------------+        +----------------+
|      size      |        |                |
+----------------+        |                |
|                |        |      data      |
|      data      |        |                |
|                |        |                |
+----------------+        +----------------+
  varying size,             varying size,
    pointers                 no pointers

+----------------+        +----------------+
|   block code   |        |   block code   |
+----------------+        +----------------+
| back reference |        |                |
+----------------+        |                |
|                |        |      data      |
|      data      |        |                |
|                |        |                |
+----------------+        +----------------+
   fixed size,               fixed size,
    pointers                 no pointers
```

Figure 2. Block Layouts.

Allocation in the heap is performed by incrementing the free space pointer hepfrp (see Figure 1) by the amount of the allocation request.

The stack grows from stkbas backwards towards hepfrp (see Figure 1); sp indicates the top of the stack. Variables and values on the stack are represented by two-cell blocks as depicted in Figure 3.

```
+----------------+
|     offset     |<----- sp
+----------------+
|      base      |
+----------------+
```

Figure 3. Stack representation of variables and values.

The meanings of base and offset are summarized in Table 1.

Table 1. Base and Offset for Variables and Values.

base	base symbol	size	offset	meaning
>0		2	<0	value is mem(base-offset)
0		2		value is offset
-1	TVNONE	2		not used
-2	TVLCL	2	<0	value is mem(cfp+offset)
-3	TVPOS	2		value is &pos
-4	TVRAND	2		value is &random
-5	TVSUBS	6	<0	value is substr(s,i,l): -offset-1 is i mem(sp+2) is value of s -mem(sp+3)-2 is l mem(sp+4) is offset for s mem(sp+5) is base for s
-6	TVSUBJ	6	<0	value is substring of &subject: -offset-1 is i mem(sp+2) is value of s -mem(sp+3)-2 is l -mem(sp+4)-1 is new &pos or mem(sp+4) is 0 if &pos is to be positioned at end of the replacement string mem(sp+5) is 0
-7	TVTBL	6	<0	value is a table element t[e]: -offset is hash bucket offset mem(sp+2) is t mem(sp+4) is e
-8	TVARRY	6	<0	value is open list element a[i]: -offset-1 is i mem(sp+4) is a

Negative values of base indicate "trapped variables"; these represent kinds of access that requires special processing [13]. Note that several of the trapped variables occupy 6 stack cells.

Since the stack is used by the reclamation routines to locate accessible data, it is essential that "junk" -- anything that is not a valid pointer -- never get pushed onto the stack. There are, however, cases in which it is necessary to push arbitrary integer data, such as return labels, onto the stack. This is done by preceding such data by -1 and a count of the number of junk cells that follow. For example, Figure 4 shows the stack configuration after pushing the junk values 52 and -104.

```
+----------------+
|           -104 |<----- sp
+----------------+
|             52 |
+----------------+
|              2 |
+----------------+
|             -1 |
+----------------+
|                |
/        .       /
/        .       /
/        .       /
|                |<----- stkbas
+----------------+
```

Figure 4. Stack Configuration After Pushing 52 and -104.

Figures depicting the representation of all types of data are given in Appendix E.

3.2 Reclamation

Reclamation of storage in each region consists of identifying the accessible data in that region, and restructuring the region so that the space occupied by inaccessible data is made available for reuse. Determining the accessible data in any region starts by examining the contents of a set of locations that may contain pointers; these locations are called _tended_ locations. The tended locations include the system stack and about two dozen specific locations in the labeled common ctend.

Reclamation in the integer region consists of "sweeping" the tended locations and the heap for integers. Prior to sweeping, a bit map of the integer region is pushed onto the stack; this is used to record accessible integers.

Reclamation in the qualifier region is similar to that in the integer region. The bit map is not needed, however. Accessible qualifiers are "marked" by setting their location fields to -(location+1) (the location field of free qualifiers is -1).

Reclamation in the string region consists of sorting pointers to the active qualifiers by their location fields, and compacting accessible strings into the lower part of the string region. The complete algorithm is given in Ref. [12].

Heap reclamation is performed using the SITBOL compactifying garbage collection algorithm [13]. There is an important aspect of this scheme that had a significant effect on the implementation. The marking phase of the algorithm constructs a linked list of pointers to every accessible block. For each such block, this list begins at its type code field (see Figure 2). One problem is that pointers are simply indices into mem and consequently pointers to the tended locations in ctend are not easily represented. In order to address the tended values, the values in ctend are copied to reserved locations at the base of the stack prior to marking, and the updated values are copied back upon completion.

The second problem is that pointers are indistinguishable from type codes, which terminate the linked lists mentioned above. Thus, the heap _must_ be positioned in mem so that all pointers into the heap have values greater than the largest type code value.

Further details concerning reclamation are given in Ref. [6].

3.3 Handling Failure

The system stack holds intermediate results. This presents a problem in the implementation of the immediate termination of an expression; it is necessary to be able to discard partially computed results that are no longer relevant. On entry to any expression that may fail (and only those expressions), the current height of the system stack is saved. When evaluation of the expression is completed (with either success or failure), the stack height is reset.

For any expression that may fail, the generated code is

```
      call xmark(MAXLABELS*p + n)
        .
        .
       <code for expression>
        .
        .
    n call xdrive
```

where p is the procedure number. The code

```
    if(signal.eq.0)goto 1
```

is emitted following every operation that may fail. In every procedure, the Fortran statement

```
    1 label=flabel
```

precedes the goto switch yard; thus transfer to the failure point is effected by transfer to label 1.

The purpose of xmark is to save the current heights of the system and control stack, the current value of flabel, and set flabel to the argument. These data are saved on the stack as depicted in Figure 5. The current heights of the system and control stacks are given by the values of marksp and markcp, respecitively. In order to avoid problems during heap reclamation, marksp and markcp are saved as _offsets_ from the base of the appropriate stack.

```
+----------------+
|   old flabel   |<----- sp
+----------------+
|   old marksp   |
+----------------+
|   old markcp   |
+----------------+
|              3 |
+----------------+
|             -1 |
+----------------+
|                |<----- new marksp
+----------------+       (an offset)
```

Figure 5. Stack after Call to xmark.

Using the current value of marksp, xdrive resets the stack heights and flabel to their saved values.

The routines xmark and xdrive are also used with generators; see Section 3.6.

3.4 Loops

The possibility of <u>break</u> and <u>next</u> appearing within loops is similar in nature to the possibility of failure in expressions. It is necessary to be able to discard partially computed results on the stack that are no longer relevant. Thus, every loop begins with

```
call xlpbeg
```

and ends with

```
call xlpend
```

The purpose of xlpbeg is similar to that of xmark; it saves the appropriate data on the system stack as illustrated in Figure 6. The variable lptop is an offset to the saved data.

```
+----------------+
| cp (as offset) |<----- sp
+----------------+
|     markcp     |
+----------------+
|     marksp     |
+----------------+
|     flabel     |
+----------------+
| cp (as offset) |
+----------------+
|     markcp     |
+----------------+
|     marksp     |
+----------------+
|     flabel     |
+----------------+
|   old lptop    |
+----------------+
|              9 |
+----------------+
|             -1 |
+----------------+
|                |<----- new lptop
+----------------+       (an offset)
```

Figure 6. Stack after Call to xlpbeg.

The purpose of xlpend is similar to xdrive; it uses lptop to restore the data saved by xlpbeg. It also insures that <u>null</u> is returned by the loop expression.

The <u>break</u> expression simply causes a transfer to the statement containing the call to xlpend (see Appendix C). The <u>next</u> expression, however, calls xnext and then transfers to the code for the first statement in the loop. The routine xnext resets cp, markcp, marksp, and flabel using the data saved in the stack at lptop. It also discards partially computed results by setting sp to lptop - 11, which was the value of sp upon entry to the loop (after the call to xlpbeg). This latter action is actually <u>incorrect</u> for <u>every</u> loops; see Section 3.6.

As an example, consider a loop of the form

```
while ... do {
   ...
   break
   ...
   next
   ...
   }
```

The general form of the generated code is

```
      call xlpbeg
23001 continue
        ...             # evaluate condition
      if(signal.eq.0)goto 23002
        ...
      goto 23002        # break
        ...
      call xnext        # next
      goto 23001
        ...
      goto 23001        # do next iteration
23002 call xlpend
```

Note that two copies of cp, markcp, marksp, and flabel are saved; this is necessary for proper execution of <u>next</u> in the <u>every</u> loop. This is described further in Section 3.6.

<u>3.5 Procedure Activation</u>

Activation records for procedures are placed on the system stack. Figure 7 shows the layout of activation records.

```
+----------------+
|  return label  |<----- sp
+----------------+
|     marksp     |
+----------------+
|     markcp     |
+----------------+
|     flabel     |
+----------------+
|     lptop      |
+----------------+
| cp (as offset) |
+----------------+
|              6 |
+----------------+
|             -1 |
+----------------+
|                |
+   last local   +
|                |
+----------------+
|                |
/       .        /
/       .        /
/       .        /
|                |
+----------------+
|                |
+   first local  +
|                |
+----------------+
|                |
+ last argument  +
|                |
+----------------+
|                |
/       .        /
/       .        /
/       .        /
|                |
+----------------+
|                |
+ first argument +
|                |
+----------------+
|    procedure   |<----- cfp
+----------------+
| old cfp(offset)|
+----------------+
```

Figure 7. Layout of Activation Records for Procedures.

The base of the current activation record is indicated by cfp. Local variables and arguments are accessed using offsets from cfp. Specifically, formal parameters are considered local variables, and the value of local variable i is found at mem(cfp-2*i).

Note that the values of local variables occupy two cells in accordance with the convention described in Section 3.1. The physical location of a local variable changes if the stack is moved. Thus, local variable i (as opposed to its value) is represented as shown in Figure 8 (see also Figure 3 and Table 1).

```
+----------------+
|      -2*i      |<----- sp
+----------------+
|       -2       |
+----------------+
```

Figure 8. Stack Representation of Local Variable i.

Note that the value of -2*i is never -1, which would confuse the reclamation routines.

3.6 Generators

When a generator is invoked, it produces its first value and then prepares for the possibility of reactivation. This preparation involves saving data related specifically to the generation of alternatives (such as the bounds and increment in the *to* generator) and partially computed results that might otherwise be consumed before the generator is reactivated. A second stack, the control stack, is used for saving these kinds of data.

The data saved for a dormant generator is taken from the system stack. The precise amount of data saved is determined by the value of marksp. Specifically, sp and marksp delimit that portion of the system stack that contains partially computed results and data relevant to the generation of alternatives. For any expression containing generators, calls to xmark and xdrive are generated as they are for expressions that may fail. The call to xmark establishes the value of marksp that is used when data is pushed onto the control stack.

A generator "prepares for reactivation" by pushing any data it may need onto the system stack and calling save. This call causes all data from sp to marksp to be pushed onto the control stack. The layout of that data is depicted in Figure 9.

```
                      .
+---------------+
|     label     |<----- cp
+---------------+
|       n       |  (number of cells saved)
+---------------+
|             2 |
+---------------+
|            -1 |
+---------------+
|               |
/  n saved data /
/ cells from the /
/  system stack /
|               |
+---------------+
```

Figure 9. Control Stack after Call to save.

The label saved on the control stack (see Figure 9) is the label of the call to the generator. As described below, this is used to transfer control back to the generator so that it can generate alternatives.

Dormant generators are activated by a call to xdrive. Subroutine xdrive performs different functions depending on whether dormant generators exist or not. If the signal is failure and there are no dormant generators, xdrive behaves as described above (see Section 3.3). If dormant generators exist upon failure, xdrive activates the most recently suspended dormant generator.

The general outline of xdrive is as follows.

```
subroutine xdrive
   if (signal == 0 & dormant generators exist) {
      label = reactivation label of dormant generator
      restore system stack from top of control stack
      }
   else {  ≠ expression succeeded or has no alternatives
      label = 0
      pop and save expression value off of system stack
      reset cp and sp from marksp and markcp
      restore marksp, markcp, & flabel from data saved by xmark
      push expression value onto system stack
      }
   return
end
```

The value of label after a call to xdrive indicates whether a generator is to be reactivated. If label is non-zero, it is the label to which control should be transferred. If label is zero, execution continues without transfer of control (note that xdrive does not change signal). Every call to xdrive is followed by the statement

```
if(label.ne.0)goto 2
```

in order to reactivate the dormant generator. Subroutine xdrive detects the presence of dormant generators by noting that the size of the control stack has increased since the last call to xmark. This is accomplished using the value of markcp set by xmark.

As mentioned above, generators save local data that is needed to compute alternatives by pushing this data onto the system stack and calling save. When the generator is reactivated, the system stack is restored to its state just prior to the call to save so that the generator may retrieve these saved data. A call to a generator results in Fortran code of the form

```
L call x????(MAXLABELS*p+L)
```

where p is the current procedure number and L is the reactivation label. In order to avoid having to write two routines for every generator (one for the initial activation and one for reactivation), the value of signal is used to determine the current phase of a generator. If the signal indicates success, the generator is being activated for the first time; it initializes itself and computes its first value. If the signal indicates failure, the generator is being reactivated; it restores its local data (now on the top of the system stack) and computes its next value. A general outline of this scheme is

```
subroutine x????(lab)
   if (signal == 1) {    ≠ initial call
      initialize local variables
      }
   else {        ≠ reactivation call
      signal = 1 ≠ assume success
      restore saved local variables
      }
   generate next value
   if (generation succeeded) {
      push local variables onto system stack
      call save(lab)
      pop local variables off system stack
      push generated value onto system stack
      }
   else
      signal = 0    ≠ generation failed
   return
end
```

Note that the reactivation label -- the argument to a generator subroutine -- is passed along to save. The initialization of local variables usually involves using additional arguments passed on the system stack. These arguments are not saved, however. The information they convey is usually stored in local variables (perhaps in a different form) and saved by calling save. The form in which the local data is saved varies from generator to generator. For example, the <u>to</u> generator simply saves the current value and the limit as integers on the stack preceded by the -1 flag and a cell count.

As an example, the generated code for the expression

```
x + 2 < (1 to 5)
```

is as follows, assuming the expression appears in procedure number 1 and MAXLABELS is 1024.

```
  call xmark(1028)       ≠ set 4 as failure label
  call xlocal(1)         ≠ push x as a variable
  call xderef            ≠ dereference to get value of x
  call xcnumr            ≠ convert value to numeric
  call xpintg(1)         ≠ push integer literal "2"
  call xadd              ≠ add value of x + 2
  call xpintg(2)         ≠ push integer literal "1"
  call xpintg(3)         ≠ push integer literal "5"
5 call xto(1029)         ≠ generate 1 to 5
  if(signal.eq.0)goto 1  ≠ failure if to is exhausted
  if(xncmp(junk).ge.0)goto 1    ≠ compare top 2 numeric values
  signal=1               ≠ reset signal after comparison
4 call xdrive            ≠ here on failure
  if(label.ne.0)goto 2   ≠ jump to reactive generator
```

Alternation (|) requires a different implementation than other generators. Consider the expression e1 | e2; if alternation were treated like other generators, the generated code would be something like

```
  <evaluate e1>
  <evaluate e2>
L call xalt(MAXLABELS*p+L)
```

The problem here is that if e1 fails, e2 would not be evaluated. In addition, e2 is evaluated "too early" -- it should only be evaluated if a subsequent expression fails. The actual code for e1 | e2 is of the form

```
L     if(signal.eq.0)goto 23001 ≠ if failure evaluate e2
      call save(MAXLABELS*p+L)  ≠ save current state
      <evaluate e1>
      goto 23002                ≠ skip over e2
23001 signal=1                  ≠ here on subsequent failure...
      <evaluate e2>             ≠  reset signal and evaluate e2
23002 continue
```

Generators are reactivated by xdrive only upon subsequent failure in the expression in which they appear. The *every* expression causes repeated activation of generators even if they succeed. Since *every* e is equivalent to

```
e & (1 = 0)
```

and *every* e1 *do* e2 is equivalent to

```
e1 & {e2; 1 = 0}
```

every is implemented by simply setting the signal to 0 before the call to xdrive, thereby causing xdrive to reactivate dormant generators.

Specifically, the generated code for the expression

```
every e1 do e2
```

is of the form

```
        call xlpbeg       # begin a loop
        call xmark(MAXLABELS*p+L)
        <evaluate e1>
        call xpop         # discard value of e1
        call xevery       # reset saved control stack height
        <evaluate e2>
        call xpop         # discard value of e2
23001   continue
        signal=0          # force failure
L       call xdrive       # reactivation generators in e1
        if(label.ne.0)goto 2
23002   call xlpend       # end of loop
```

The sole purpose of xevery is to reset the second copy of cp, markcp, marksp, and flabel saved by xlpbeg (see Figure 6). The values of these variables saved by xlpbeg do not reflect the data accumulated by the evaluation of e1. If a <u>next</u> expression appears in e2, the failure label, system stack, and control stack must be restored to the state that they had immediately following the evaluation of e1. The second copy of this data saved by xlpbeg is used for this purpose and xevery simply records the current values of cp, markcp, marksp, and flabel after every activation of e1.

As mentioned in the previous section, however, xnext resets sp, which amounts to discarding partially computed results that may be required for computing alternatives of e1. The reason this action does not cause problems is that upon return from xnext, control is transferred to the statement labeled 23001 (see above), signal is set to 0, and xdrive is called. If e1 has more alternatives, xdrive resets sp from the current value of marksp, which was reset by xevery. Thus, resetting sp in xnext has no effect in <u>every</u> loops. A better technique would be to treat <u>every</u> loops differently from the other loop contructs. In particular, the duplicate copy of the data saved by xlpbeg (see Figure 6) is needed only for <u>every</u> loops, and most of the complications to <u>break</u> and <u>next</u> are caused by the generative aspects of <u>every</u>.

4. Programming Conventions and Peculiarities

There are a number of less-than-obvious programming conventions and peculiarities in the implementation of Icon that deserve discussion. Most of these are due to the use of Fortran as an implementation language.

4.1 Stability

All routines in the implementation are classified as either <u>stable</u> or <u>unstable</u>. An unstable routine is one that has the potential of causing a reclamation in some region; a stable routine is one that can never cause a reclamation. A routine is unstable if it calls an unstable routine.

This convention is very important because failure to place pointers in tended locations prior to calling unstable routines leads to time- and data-dependent bugs that are very difficult to locate. Thus, the usual technique is to push local variables that contain pointers onto the system stack before calling an unstable routine and to pop them off upon return.

4.2 Calling Unstable Routines

Actual arguments to Fortran subroutines and functions are passed by reference. This mechanism has induced a particular programming convention for calling unstable routines.

To illustrate the problem, consider the call

```
z = cat(mem(sp), mem(sp+2))
```

which appears to concatenate the two values at the top of the stack and assign the result to z. The problem is that if cat is unstable (assume it is), the value of sp may change midway during its execution due to a reclamation. If this occurs, the addresses of the actual arguments are no longer valid.

In implementations of Fortran that use call by reference for all arguments (e.g. CDC FTN [15]), references to the actual arguments after the reclamation access what appears to be junk. Some implementations of Fortran (e.g. DEC Fortran-10 [16]), use call by value-result for scalar arguments. In this kind of argument transmission, the value of the actual argument is fetched upon entry to the routine and stored in a local variable. Upon exit, the final value of the local varible is stored in the actual argument. After the reclamation in this case, not only do references to the actual argument access junk, but the assignment upon exit overwrites the wrong data in mem.

These problems are avoided by writing the above call as

```
x = mem(sp)
y = mem(sp+2)
z = cat(x, y)
```

and _not_ using the values of x and y after the call to cat.

Similar comments apply to statements such as

```
call f(g(x), y)
```

where g(x) returns a pointer and f is unstable. In this case, the value of g(x) is stored in an untended temporary location. Note, however, that this statement is safe if f copies its arguments to tended locations upon entry; this "trick" is not used.

As a result of these problems, there are very few routines that have calling sequence like cat above. Most unstable routines do not take tended arguments as Fortran arguments, but use the system stack for the transmission of tended arguments and results.

4.3 The Stack

Since the stack grows backwards, the sequence to push x onto the stack is

```
sp = sp - 1
mem(sp) = x
```

This code does not check for overflow, however. Overflow has occurred when the stack pointer (sp) equals the heap free pointer (hepfrp). Thus, the correct sequence to push x is

```
if (sp == hepfrp)
   ... overflow ...
sp = sp - 1
mem(sp) = x
```

This sequence is tedious, and is avoided by the use of stkchk. The call stkchk(n) insures that there are at least n cells of available stack space. Thus, to push several values onto the stack, stkchk is called to insure enough room followed by code to push each value. For example, the following code pushes x and y onto the stack as junk.

```
call stkchk(4)
mem(sp-1) = -1
mem(sp-2) = 2
mem(sp-3) = x
mem(sp-4) = y
sp = sp - 4
```

The pushes are written in this fashion to avoid repeated decrements of sp. In a reference like mem(sp-3), most Fortran compilers will absorb the -3 into the address portion of the instruction. As a result, the above sequence usually generates good code using the value of sp (in a register) as an offset from 4 different addresses.

As shown in Appendix C, there are no calls to stkchk in the generated code. The translator estimates the amount of stack space required to execute a procedure, and stkchk is called during procedure invocation to insure sufficient stack space.

Acknowledgements

Icon was designed by Ralph Griswold, David Hanson, and Tim Korb. Cary Coutant, Walter Hansen, and Steve Wampler made valuable suggestions. The version of Icon described in this report was implemented by the authors and Tim Korb. Some of the material in this report, especially that concerning the implementation of generators, was adapted from of Tim Korb's PhD dissertation [5].

References

1. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1971.

2. R. E. Griswold and D. R. Hanson, "An Overview of SL5", SIGPLAN Notices 12, 40-50 (1977).

3. R. E. Griswold, D. R. Hanson and J. T. Korb, "The Icon Programming Language: An Overview", SIGPLAN Notices 14, 18-31 (1979).

4. R. E. Griswold and D. R. Hanson, Reference Manual for the Icon Programming Language, Tech. Rep. TR79-1a, Dept. of Comp. Science, Univ. Arizona, Tucson, 1980.

5. John T. Korb, The Design and Implementation of a Goal-Directed Programming Language, PhD Dissertation, Tech. Rep. TR79-11, Dept. of Comp. Science, Univ. Arizona, Tucson, 1979.

6. D. R. Hanson, "A Portable Storage Management System for the Icon Programming Language", Software -- Practice and Experience. to appear (1980).

7. A. V. Aho and J. D. Ullman, Principles of Compiler Design. Addison-Wesley, Reading, MA, 1977.

8. N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, NJ, 1971, Chap. 5.

9. C. J. Prenner, J. M. Spitzen, and B. Wegbreit, "An Implementation of Backtracking for Programming Languages", SIGPLAN Notices 7, 36-44 (1972).

10. D. G. Bobrow and B. Wegbreit, "A Model and Stack Implementation of Multiple Environments", Communications of the ACM 16, 591-603 (1973).

11. D. R. Hanson, "A Simple Technique for Representing Strings in Fortran IV", Communications of the ACM 17, 646-647 (1974).

12. D. R. Hanson, The Manipulation of Varying-Length String Data in Fortran IV, Tech. Rep., Dept. of Comp. Science, Univ. Arizona, Tucson, 1975.

13. D. R. Hanson, "Storage Management for an Implementation of SNOBOL4," Software -- Practice and Experience 7, 179-192 (1977).

14. D. R. Hanson, "Variable Associations in SNOBOL4," Software -- Practice and Experience 6, 245-254 (1976).

15. Fortran Extended Reference Manual, Control Data Corp., Sunnyvale, CA, 1978.

16. Fortran-10 Language Manual, Order code DEC-10-LFORA-B-D, Digital Equipment Corp., Maynard, MA, 1974

17. B. G. Ryder, "The PFORT Verifier," Software -- Practice and Experience 4, 359-377 (1974).

Appendix A. Format of the Generated Icon Program

The following code is an example of the overall structure of the Fortran code generated by the translator.

```
      subroutine icon
      common /cmain/signal,label,flabel,line
      integer signal,label,flabel,line
      integer xcmp,xlcmp,xncmp,xcomp
      integer s(NS+1), p(NP+1), g(NG+1), i(NI+1), l(NL+1)
      real r(NR+1)
      data s/NS, .../
      data g/NG, .../
      data p/NP, .../
      data i/NI, .../
      data l/NL, .../
      data r/NR, .../
      call sinit(s,g,p,i,r,l)
      call xglobl(1)          ≠ 1 = main procedure global
      call xderef             ≠ get the procedure value
      call xcproc             ≠ convert to procedure and go to it
      call xinvok((NP+1)*MAXLABELS,0)
      goto 23001
    1 call p1
      goto 23001
        .
        .
        .
    i call pi
      goto 23001
        .
        .
        .
   NP call pNP
      goto 23001
 NP+1 return
23001 kk=label/MAXLABELS
      goto (1,2,...,NP),kk
      call syserr(29hicon: illegal internal label.)
      return
      end
        .
        .
        .
      subroutine pi           ≠ one for each Icon procedure
      common /cmain/signal,label,flabel,line
      integer signal,label,flabel,line
      integer xcmp,xlcmp,xncmp,xcomp
      goto 2
    3 continue
       .
       .                      ≠ generated code (see Appendix B)
       .
    1 label=flabel
```

```
50      2 if(label/MAXLABELS .ne. i)return
51        kk=mod(label,MAXLABEL)
52        goto(1,2,3,...,n),kk
53        call syserr(27hpi: illegal internal label.)
54        return
55        end
```

The numbers in the following explanation refer to line numbers in the above program skeleton.

2-3, 41-42: The labeled common cmain contains the global identifiers referenced by the compiled code; it serves to communicate current status information to and from the runtime system. Signal is the current value of the signal, label is used for transfer of control much like a location counter, flabel contains the label to which control is transferred upon failure, and line is the line number in the source program of the current focus of execution.

4,43: These routines are called for comparisons; they return integers.

5-12: These arrays contain initialization data. The general format is that the first element contains a count of the number of data elements that follow. The array s contains the characters appearing in string literals and identifiers. Each string is terminated by an EOS character. The array p contains control data for each procedure; g contains global identifier names as indices into s; i contains literal integers; l contains literal strings as indices into s; and r contains literal reals.

13: Sinit initializes storage and copies the data appearing in s, g, p, i, r, and l into the appropriate Icon storage regions.

14-18: This is code to invoke the main procedure much in the same way that other procedures are invoked. The return point, labeled NP+1 where NP is the number of procedures, causes a return to the Fortran main program, which terminates execution.

19-23: Labels that may be targets for transfer of control are composed of two parts, a procedure number p and an internal (Fortran) label i. A label is represented by p*MAXLABELS+i. Whenever control leaves a procedure, it returns to the subroutine icon, which serves to transfer control to another procedure. The code in this section accomplishes this kind of transfer.

40,44-45: Each Icon procedure is translated into a Fortran subroutine named pi, where i is the procedure number. The procedure entry point is in line 44, but execution begins by transferring to the local "switch yard" in case the procedure was suspended.

46-48: Generated code for the procedures as described in Appendix B.

49-52: This is the local goto "switch yard" that is used to transfer control. The label 1 causes the current value of flabel to be used; transfer of control upon failure is effected by transfer to this label. If the target is not in the current procedure, a return to subroutine icon is made, otherwise control is transferred as indicated by the local label portion of the value of label.

With two exceptions, the Fortran code generated by the Icon translator conforms to the Fortran standard as embodied in the PFORT verifier [17].

The most serious exception is in the data statements illustrated in lines 7-12 above. It is assumed that arrays can be initialized by data statements of the form

```
data a /value of a(1), value of a(2), ..., value of a(N)/
```

or, if the array is large, by data statements of the form

```
data (a(k),k=1,100) /value of a(1), ..., value of a(100)/
data (a(k),k=101,200) /value of a(101), ..., value of a(200)/
data (a(k),k=201,N) /value of a(201), ..., value of a(N)/
```

The ANSI standard form requires explicit specification of each element of the array, i.e.

```
data a(1) /value of a(1)/
data a(2) /value of a(2)/
   ...
data a(N) /value of a(N)/
```

If the form of the data statements causes problems, it can be changed by modifying the translator routine outds, which is called to output most data statements. Exceptions are the array r (see line 12 above), which is output in outhdr, and field offset arrays, which are described below.

The second exception concerns the use of block data subprograms. If records are used in an Icon program, arrays containing field offsets are generated and placed the labeled common cflds. A block data subprogram is generated that initializes these arrays. The problem is that, in this case, there are two block data subprograms -- the one that initializes cflds and one that initializes other runtime data used by every Icon program. Having more than one block data subprogram is contrary to the ANSI standard and may cause problems. If so, the offset arrays can be made local to each Fortran procedure (corresponding to each Icon procedure) by modifying the translator routine outfld. This routine is called to output the common statement in each subroutine and may be modified to output the data statements in place of the common statement. Note that the arrays are output directly by outfld; outds is not called. Thus, if the form of data statement mentioned above causes problems, outfld will need to be modified.

Appendix B. A Grammar for Icon

The following grammar for Icon is left-factored and has no left-recursive productions. The code segments in the parser for each non-terminal are derived from the productions in this grammar. Language constructs processed by pass one, such as record and global declarations, are not shown.

```
PROC     -->  procedure INIT SLIST
INIT     -->  initial DEXP
            | eps
SLIST    -->  DEXP SLISTP
SLISTP   -->  ';' DEXP SLISTP
            | <newline> DEXP SLISTP
            | eps
DEXP     -->  EXP
EXP      -->  AND
            | eps
AND      -->  ASSIGN ANDP
ANDP     -->  '&' ASSIGN ANDP
            | eps
ASSIGN   -->  TOBY ATYP
ATYP     -->  ":=" ASSIGN
            | ":=:" ASSIGN
            | "<-" ASSIGN
            | "<->" ASSIGN
            | eps
TOBY     -->  OR TOBYP
TOBYP    -->  to OR BY TOBYP
            | eps
BY       -->  by OR
            | eps
OR       -->  RELOP ORP
ORP      -->  '|' RELOP ORP
            | eps
RELOP    -->  CONCAT RELOPP
RELOPP   -->  '='    CONCAT RELOPP
            | "~="   CONCAT RELOPP
            | '<'    CONCAT RELOPP
            | "<="   CONCAT RELOPP
            | '>'    CONCAT RELOPP
            | ">="   CONCAT RELOPP
            | "=="   CONCAT RELOPP
            | "~=="  CONCAT RELOPP
            | "==="  CONCAT RELOPP
            | "~===" CONCAT RELOPP
            | eps
CONCAT   -->  ADDOP CONCATP
CONCATP  -->  "||" ADDOP CONCATP
            | eps
ADDOP    -->  MULOP ADDOPP
ADDOPP   -->  '+' MULOP ADDOPP
            | '-' MULOP ADDOPP
            | "++" MULOP ADDOPP
            | "--" MULOP ADDOPP
```

```
                | eps
MULOP    -->    EXPOP MULOPP
MULOPP   -->    '*' EXPOP MULOPP
                | '/' EXPOP MULOPP
                | "**" EXPOP MULOPP
EXPOP    -->    SUFFIX ETYP
ETYP     -->    '^' EXPOP
                | eps
SUFFIX   -->    PREFIX SUFFIXP
SUFFIXP  -->    '+' SUFFIXP
                | '-' SUFFIXP
                | fails SUFFIXP
                | eps
PREFIX   -->    '~' PREFIX
                | '+' PREFIX
                | '-' PREFIX
                | '=' PREFIX
                | '!' PREFIX
                | PRIME
PRIME    -->    '&' KEYWORD PRIMEP
                | LOCAL PRIMEP
                | GLOBAL PRIMEP
                | BUILTIN '(' ELIST ')' PRIMEP
                | RNAME '(' ELIST ')' PRIMEP
                | LITERAL PRIMEP
                | '(' EXP ')' PRIMEP
                | '<' ELIST '>' PRIMEP
                | '{' SLIST '}' PRIMEP
                | if DEXP then DEXP ELSEX PRIMEP
                | while DEXP do DEXP PRIMEP
                | until DEXP do DEXP PRIMEP
                | every EXP DOX PRIMEP
                | repeat DEXP PRIMEP
                | fail PRIMEP
                | succeed RETX PRIMEP
                | return RETX PRIMEP
                | suspend RETX PRIMEP
                | break PRIMEP
                | next PRIMEP
                | stack '(' EXP ')' PRIMEP
                | table '(' EXP ')' PRIMEP
                | list '(' PROTO ')' AINIT PRIMEP
                | scan DEXP using DEXP PRIMEP
                | case DEXP of '{' CLIST '}' PRIMEP
PRIMEP   -->    '(' ELIST ')' PRIMEP
                | '{' EXP '}' PRIMEP
                | '.' FNAME PRIMEP
                | eps
ELSEX    -->    else DEXP
                | eps
DOX      -->    do DEXP
                | eps
RETX     -->    DEXP
                | eps
```

```
ELIST    -->  EXP ELISTP
ELISTP   -->  ',' EXP ELISTP
            | eps
LITERAL  -->  INT
            | FLOAT
            | STRING
ALIST    -->  LOCAL ALISTP
ALISTP   -->  ',' LOCAL ALISTP
            | eps
PROTO    -->  DEXP UBX
            | eps
UBX      -->  ':' DEXP
            | eps
AINIT    -->  initial DEXP
            | eps
CLIST    -->  CELEM CLISTP
CLISTP   -->  ';' CELEM CLISTP
            | <newline> CELEM CLISTP
            | eps
CELEM    -->  default ':' DEXP
            | LLIST ':' DEXP
LLIST    -->  LITERAL LLISTP
LLISTP   -->  ',' LITERAL LLISTP
            | eps
```

Appendix C. Syntax and Corresponding Fortran Code

This appendix gives an informal BNF-like description of Icon and the form of the corresponding generated code. The notation is as follows: curly braces denote required constructs, square brackets denote optional constructs, and ellipses following a group denote repetition. No attempt has been made to show relative precedence in expressions. A nonterminal appearing in the code denotes its own generated code.

A <procedure> is

<u>procedure</u> <ident> [<header-decl>] <dexp1>... <u>end</u>

```
L1  continue                       ≠ generate entry label
    call xreset(L2)                ≠ reset if initial
    <dexp0>                        ≠ evaluate initial clause
L2  continue                       ≠ secondary entry point
    <dexp1>                        ≠ generate procedure body
    call xpop                      ≠ pop final value
    call xpnull                    ≠ generate default return
    call xretrn                    ≠ and return
    goto JUMP
```

A <dexp> is

<u>if</u> <dexp> <u>then</u> <dexp1> <u>else</u> <dexp2>

```
    <dexp>                         ≠ evaluate boolean
    call xpop                      ≠ throw away value
    if (signal .eq. 0) goto F1     ≠ jump if <dexp> failed
    <dexp1>                        ≠ evaluate the statment
    goto F2                        ≠ skip second expression
F1  signal = 1                     ≠ reset signal to success
    <dexp2>                        ≠ evaluate second expression
F2  continue
    if (signal .eq. 0) goto FAIL   ≠ check failure if necessary
```

<u>while</u> <dexp> <u>do</u> <dexp>

```
                                   ≠ F1 = next, F2 = break lab
    call xlpbeg                    ≠ establish loop beginning
F1  <dexp>                         ≠ evaluate boolean
    call xpop                      ≠ discard its value
    if (signal .eq. 0) goto F2     ≠ skip out on failure
    <dexp>                         ≠ evaluate the expression
    call xpop                      ≠ discard its value
    goto F1                        ≠ and loop
F2  call xlpend                    ≠ close down loop
```

until <dexp> do <dexp>

```
    call xlpbeg                       ≠ establish loop beginning
F1  <dexp>                            ≠ evaluate boolean
    call xpop                         ≠ discard its value
    if (signal .eq. 1) goto F2        ≠ skip out on success
    signal = 1                        ≠ remove failure signal
    <dexp>                            ≠ evaluate the expression
    call xpop                         ≠ discard its value
    goto F1                           ≠ and loop
F2  call xlpend                       ≠ close down loop
```

every <exp>

```
    call xmark(L1)                    ≠ mark stacks; failure label
    <exp>                             ≠ evaluate expression
    call xpop                         ≠ throw away value
    signal = 0                        ≠ force failure
L1  call xdrive                       ≠ iterate
    if (label .ne. 0) goto JUMP       ≠ jump if more alternatives
    signal = 1                        ≠ loop must succeed
```

every <exp> do <dexp>

```
    call xlpbeg                       ≠ establish loop beginning
    call xmark(L1)                    ≠ mark stacks; failure label
    <exp>                             ≠ evaluate the generator
    call xpop                         ≠ throw away value
    call xevery                       ≠ mark c stack data
    <dexp>                            ≠ evaluate the expression
    call xpop                         ≠ throw away value
F1  continue                          ≠ next label
    signal = 0                        ≠ force failure
L1  call xdrive                       ≠ iterate
    if (label .ne. 0) goto JUMP       ≠ jump if more alternatives
F2  call xlpend                       ≠ close down loop
```

repeat <dexp>

```
    call xlpbeg                       ≠ establish loop beginning
F1  <dexp>                            ≠ evaluate expression
    call xpop                         ≠ discard value
    if (signal .eq. 1) goto F1        ≠ loop on success
F2  call xlpend                       ≠ close down loop
```

scan <dexp1> using <dexp2>

```
    <dexp1>                           ≠ evaluate subject
    if (signal .eq. 0) goto FAIL      ≠ abandon on failure
    call xderef                       ≠ dereference if needed
    call xcstrg                       ≠ convert to string if needed
    call xscan1                       ≠ setup for scanning
    <dexp2>                           ≠ do scanning
    call xscan2                       ≠ restore &subject and &pos
    if (signal .eq. 0) goto FAIL      ≠ check for failure
```

case <dexp0> of { [{ <liti>, }... | default : <dexpj>]... }

```
    <dexp0>                           ≠ evaluate case expression
    call xecase                       ≠ error if <dexp0> fails
    if (xcomp(n1,t1).ne.0) goto F2    ≠ n1 = literal, t1 = type
    call xpop                         ≠ discard case expression value
    <dexp1>                           ≠ evaluate expression for lit1
    goto F1                           ≠ skip remainder of case
F2  if (xcomp(n2,t2).ne.0 .and.
        xcomp(n3,t3).ne.0)) goto F3   ≠ check next set of literals
    call xpop                         ≠ discard case expression value
    <dexp2>                           ≠ evaluate expression for lit2
    goto F1                           ≠ skip remainder of case
      .
      .
      .
Fn  call xpop                         ≠ discard case expr value
    <default expression>              ≠ evaluate default
F1  continue
```

fail

```
    call xpnull                       ≠ push &null
    signal = 0                        ≠ force failure
    call xretrn                       ≠ return &null,failure
    goto JUMP                         ≠ computed branch
```

succeed

```
    call xpnull                       ≠ push &null
    signal = 1                        ≠ force success
    call xretrn                       ≠ return &null,succeed
    goto JUMP                         ≠ computed branch
```

succeed <dexp>

```
    <dexp>                              ≠ evaluate return value
    signal = 1                          ≠ force success
    call xretrn                         ≠ return value,success
    goto JUMP                           ≠ computed branch
```

return

```
    call xpnull                         ≠ push &null
    signal = 1                          ≠ force success
    call xretrn                         ≠ return &null,success
    goto JUMP                           ≠ computed branch
```

return <dexp>

```
    <dexp>                              ≠ evaluate return value
    call xretrn                         ≠ return value,signal
    goto JUMP                           ≠ computed branch
```

suspend

```
    call xpnull                         ≠ push &null
L1  call xsusp(L1)                      ≠ set up for suspend &null
    if (label .ne. 0) goto JUMP         ≠ jump if really suspending
    signal = 1                          ≠ insure success signal
```

suspend <exp>

```
    call xmark(L1)                      ≠ mark stacks
    <exp>                               ≠ evaluate the argument
L2  call xsusp(L2)                      ≠ suspend argument
    if (label .ne. 0) goto JUMP         ≠ jump if really suspending
L1  call xdrive                         ≠ extract alternative value
    if (label .ne. 0) goto JUMP         ≠ jump if we have any
    signal = 1                          ≠ force success
```

break

```
    goto F2                             ≠ break from loop (to xlpend)
```

next

```
    call xnext                          ≠ adjust stack heights
    goto F1                             ≠ iterate
```

< [<exp> ,]... >

```
    <expl>                          ≠ compute first element
    call xderef                     ≠ dereference if necessary
    <exp2>                          ≠ compute second element
    call xderef                     ≠ dereference if necessary
        .
        .
        .
    call xllist(n)                  ≠ n = number of elements
```

<dexp>

```
    call xmark(L1)                  ≠ mark stacks; failure label
    <exp>                           ≠ evaluate expression
L1  call xdrive                     ≠ drive expression to success
    if (label .ne. 0) goto JUMP     ≠ jump if more alternatives
```

<exp> is

For built-in prefix, suffix, and infix operators and built-in functions, the code sequences given below are the maximum that may be required. In general, the label L1 (both in label position and as argument) is supplied only if the operation is a generator. The signal test and branch to FAIL is eliminated for unconditional operations (note that all generators are conditional). Dereferencing and conversion code is also optional. Dereferencing is required whenever a variable (either natural or computed, including values returned by defined procedures) is given when a value is required. A call to a conversion routine is required when an argument is of the wrong or unknown type. The conversion routines are

```
xcintg  convert to integer
xcstrg  convert to string
xcfile  convert to file
xcreal  convert to real
xccset  convert to character set
xcnumr  convert to numeric
xcproc  convert to procedure
xcrecd  convert to record
```

<prefix> <exp>

```
    <exp>                           ≠ evaluate expression
    call xderef                     ≠ dereference arg if needed
    call xc????                     ≠ convert arg if needed
L1  call "opcode"(L1)               ≠ "opcode" is prefix name
    if (signal .eq. 0) goto FAIL    ≠ check failure if needed
```

<exp> <suffix>

```
    <exp>                            ≠ evaluate expression
    call xderef                      ≠ dereference arg if needed
    call xc????                      ≠ convert arg if needed
L1  call "opcode"(L1)                ≠ "opcode" is suffix name
    if (signal .eq. 0) goto FAIL     ≠ check failure if needed
```

<exp1> <infix> <exp2>

```
    <exp1>                           ≠ evaluate first operand
    call xderef                      ≠ dereference arg if needed
    call xc????                      ≠ convert arg if needed
    <exp2>                           ≠ evaluate second operand
    call xderef                      ≠ dereference arg if needed
    call xc????                      ≠ convert arg if needed
L1  call "opcode"(L1)                ≠ "opcode" is infix name
    if (signal .eq. 0) goto FAIL     ≠ check failure if needed
```

The code for numeric and lexical comparisons uses two functions, xncmp and xlcmp. These functions compare the top two values on the stack and return -1, 0, or +1, if the top value is less than, equal to, or greater than the value below it on the stack. This result is then compared to zero using one of the Fortran comparisons (e.g., .le.).

<exp1> <relop> <exp2>

```
    <exp1>                           ≠ evaluate first argument
    call xderef                      ≠ dereference arg if needed
    call xc????                      ≠ convert arg if needed
    <exp2>                           ≠ evaluate second operand
    call xderef                      ≠ dereference arg if needed
    call xc????                      ≠ convert arg if needed
    if (x?cmp(junk).??.0) goto FAIL  ≠ check relation
```

<ident> ([<exp> ,]...)

```
                                     ≠ built-in procedure
    <exp1>                           ≠ evaluate first argument
    call xderef                      ≠ dereference arg if needed
    call xc????                      ≠ convert arg if needed
    <exp2>                           ≠ evaluate second argument
    call xderef                      ≠ dereference arg if needed
    call xc????                      ≠ convert arg if needed
     .
     .
     .
L1  call "opcode"(L1)                ≠ pass correct number of args
    if (signal .eq. 0) goto FAIL     ≠ check failure
```

```
<expl> ( [ <exp2> , ]... )

                                        ≠ procedure call
    <expl>                              ≠ evaluate proc name
    call xderef                         ≠ dereference proc
    call xcproc                         ≠ convert to procedure
    <exp2>                              ≠ evaluate first arg
    call xderef                         ≠ dereference arg if needed
    <exp3>                              ≠ evaluate second arg
    call xderef                         ≠ dereference arg if needed
     .
     .
     .
    call xinvok(L1,n)                   ≠ n = number of args
    goto JUMP                           ≠ jump to the procedure
L1  if (signal .eq. 0) goto FAIL        ≠ check failure
```

```
{ [ <dexp> ]... }

    <dexp>                              ≠ generate expression code
    call xpop                           ≠ discard value (except last)
```

```
<exp0> [ <expl> ]

    <exp0>                              ≠ evaluate list or string
    <expl>                              ≠ evaluate subscript
    call xacc                           ≠ access list
    if (signal .eq. 0) goto FAIL        ≠ check failure
```

```
<expl> & <exp2>

    <expl>                              ≠ evaluate left argument
    call xpop                           ≠ throw away value
    <exp2>                              ≠ evaluate second argument
```

```
<expl> | <exp2>

L1  if (signal .eq. 0) goto F1          ≠ F1 = alternate label
    call save(L1)                       ≠ L1 = reactivation label
    <expl>                              ≠ evaluate first expression
    goto F2                             ≠ skip second expression
F1  signal = 1                          ≠ reset for success
    <exp2>                              ≠ evaluate second expression
F2  continue
```

<exp1> to <exp2>

```
    <exp1>                          ≠ evaluate from expression
    call xderef                     ≠ dereference arg if needed
    call xcintg                     ≠ convert to integer
    <exp2>                          ≠ evaluate to expression
    call xderef                     ≠ dereference arg if needed
    call xcintg                     ≠ convert to integer
L1  call xto(L1)                    ≠ generate values
    if (signal .eq. 0) goto FAIL    ≠ check failure
```

<exp1> to <exp2> by <exp3>

```
    <exp1>                          ≠ evaluate from expression
    call xderef                     ≠ dereference arg if needed
    call xcintg                     ≠ convert to integer
    <exp2>                          ≠ evaluate to expression
    call xderef                     ≠ dereference arg if needed
    call xcintg                     ≠ convert to integer
    <exp3>                          ≠ evaluate by expression
    call xderef                     ≠ dereference arg if needed
    call xcintg                     ≠ convert to integer
L1  call xtoby(L1)                  ≠ generate values
    if (signal .eq. 0) goto FAIL    ≠ check failure
```

<dexp> fails

```
    <dexp>                          ≠ evaluate expression
    signal = iabs(signal - 1)       ≠ invert signal
    if (signal .eq. 0) goto FAIL    ≠ check failure
```

list ([<exp1> :] <expu>) [initial <exp>]

```
    <exp1>                          ≠ evaluate lower bound
    <expu>                          ≠ evaluate upper bound
    <exp>                           ≠ evaluate initial value
    call xmarry(0 or 1)             ≠ make list
```

table (<exp>)

```
    <exp>                           ≠ evaluate table size
    call xmtabl                     ≠ make the table
```

stack (<exp>)

```
    <exp>                           ≠ evaluate stack size
    call xmstak                     ≠ make the stack
```

```
<record name> ( <expl>, ..., <expm> )

    <expl>                          ≠ evaluate first field
        .
        .
        .
    <expm>                          ≠ evaluate last field
    call xmrecd(t,k,m,n)            ≠ make record, t = record type,
                                    ≠ k = record name,
                                    ≠ n = number of fields


<exp> . <identifier>

    <exp>                           ≠ evaluate record expression
    call xfacc(fi)                  ≠ fi = field offset array


<integer>

    call xpintg(n)                  ≠ n = integer offset


<real>

    call xpreal(n)                  ≠ n = real offset


<string>

    call xpstrg(n)                  ≠ n = string offset


<local identifier>

    call xlocal(n)                  ≠ n = identifier offset


<global identifier>

    call xglobl(n)                  ≠ n = identifier offset
```

A <body> is

```
[ <decl> ]... [ <dexp> ]...
```

A <decl> is

<u>local</u> [<ident> ,]...

A <literal> is

```
" string not containing double quote "
' string not containing single quote '
{ digit }...
{ digit }... . [ digit ]...
```

A <ident> is

```
{ alpha } [ alpha | digit | underscore { alpha | digit }...]...
```

Appendix D. Built-in Operators, Functions, and Keywords

This appendix contains a list of the Icon built-in operations. The following information is included with each operation.

- argument information
- type of the returned result
- failure indication
- generator indication

The argument information is either the datatype expected or the default if no argument is given (from which the type may be deduced). Note that all generators may fail.

C.1 Built-in Operators

```
+( -- convert to numeric          call xnumr
+any -> numeric


-( -- negate                      call xneg
-numeric -> numeric


=( -- tab over matched string     call xtabm       generator
=string -> string


~( -- negate character set        call xnotc
~cset -> cset


!( -- access structure            call xbang       generator
!any -> variable


)+ -- increment                   call xdupl
                                  call xderef
                                  call xcnumr
                                  call xpone
                                  call xadd
variable+ -> variable


)- -- decrement                   call xdupl
                                  call xderef
                                  call xcnumr
                                  call xpone
                                  call xsub
variable- -> variable
```

```
)^( -- power                          call xpower
numeric ^ numeric -> numeric


)*( -- multiplication                 call xmul
numeric * numeric -> numeric


)+( -- addition                       call xadd
numeric + numeric -> numeric


)-( -- subtraction                    call xsub
numeric - numeric -> numeric


)/( -- division                       call xdiv
numeric / numeric -> numeric


)**( -- character set union           call xunion
cset ** cset -> cset


)--( -- character set difference      call xdiff
cset -- cset -> cset


):=( -- assignment                    call xasg          may fail
variable := any -> any


):=:( -- value swap                   call xswap         may fail
variable :=: variable -> any


)<( -- .lt. predicate                 xncmp(junk)
numeric < numeric -> numeric


)<=( -- .le. predicate                xncmp(junk)
numeric <= numeric -> numeric


)=( -- .eq. predicate                 xncmp(junk)
numeric = numeric -> numeric


)==( -- .eq. string                   xlcmp(junk)
string == string -> string
```

```
)>( -- .gt. predicate              xncmp(junk)
numeric > numeric -> numeric

)>=( -- .ge. predicate             xncmp(junk)
numeric >= numeric -> numeric

)||( -- concat                     call xcat
string || string -> string

)~=( -- .ne. predicate             xlcmp(junk)
numeric ~= numeric -> numeric

)~==( -- .ne. string               xlcmp(junk)
string ~== string -> string

)===( -- .eq. structure            xcmp(junk)
any === any -> any

)~===( -- .ne. structure           xcmp(junk)
any ~=== any -> any

)<-( -- reversible assignment      call xrasg      generator
variable <- any -> any

)<->( -- reversible swap           call xrswap     generator
variable <-> variable -> any
```

C.2 Built-in Functions

```
any -- match character             call xany       may fail
any(cset:"",string:&subject,
       integer:&pos or 1,integer:0) -> integer

bal -- match balanced string       call xbal       generator
bal(cset:&ascii,cset:"(",cset:")",
       string:&subject,integer:&pos or 1,integer:0) -> integer

center -- center text in string    call xcent
center(string:"",integer:0,string:" ") -> string
```

```
close -- close object               call xclose
close(any:file,table,or array) -> argument


copy -- copy structure              call xcopy
copy(any:null) -> argument


cset -- convert to character set    call xcset
cset(any:null) -> cset


display -- display symbol table     call xdisp
display(integer:1) -> null


find -- find string                 call xfind       generator
find(string:"",string:&subject,
        integer:&pos or 1,integer:0) -> integer


image -- convert to string image    call ximage
image(any:null) -> string


integer -- convert to integer       call xnumr       may fail
integer(any:null) -> integer        if (signal .eq. 0) go to FAIL
                                    call xintg


left -- left justify in string      call xleft
left(string:"",integer:0,string:" ") -> string


lge -- lexical >=                   call xlge        may fail
lge(string:"",string:"") -> string


lgt -- lexical >                    call xlgt        may fail
lgt(string:"",string:"") -> string


lle -- lexical <=                   xlcmp(junk)      may fail
lle(string:"",string:"") -> string


llt -- lexical <                    xlcmp(junk)      may fail
llt(string:"",string:"") -> string
```

```
many -- span characters              call xmany      may fail
many(cset:"",string:&subject,integer:&pos or 1,
                                integer:0) -> integer


map -- translate characters          call xmap
map(string:"",string:"",string:"") -> string


match -- match string                call xmatch     may fail
match(string:"",string:&subject,integer:&pos or 1,
         integer:0) -> integer


mod -- get remainder                 call xmod
mod(numeric:0,numeric:0) -> numeric


move -- move in &subject             call xmove      generator
move(integer:0) -> variable


null -- check for null               call xnull      may fail
null(any:null) -> null


numeric -- convert to numeric        call xnumr      may fail
numeric(any:null) -> numeric


open -- open object                  call xopen      may fail
open(any:file,table,or array,string:"") -> argument


pop -- pop off stack                 call xpops      may fail
pop(stack) -> any


pos -- convert to cursor position    call xpos       may fail
pos(integer:0,string:&subject) -> integer


push -- push onto stack              call xpushs
push(stack, any) -> any


random -- compute random integer     call xrand
random(integer:1) -> integer


read -- read line                    call xread      may fail
read(file:&input) -> string
```

```
reads -- read string                call xsread     may fail
reads(file:&input,integer:1) -> string


real -- convert to real             call xcreal     may fail
real(any:null) -> real


repl -- replicate string            call xrepl
repl(string:"",integer:0) -> string


reverse -- reverse string           call xrev
reverse(string:"") -> string


right -- right justify in string    call xright
right(string:"",integer:0,string:" ") -> string


section -- get section              call xsect      may fail
section(variable:&subject,integer:&pos or 1,integer:0) -> variable


size -- size of an object           call xsize
size(any:string or list) -> integer


sort -- sort array or table         call xsort      may fail
sort(any:null,integer:1) -> argument


stop -- stop execution              call xstop(n)
stop(any list) -> null              (n = number of arguments)


string -- convert to string         call xstrg      may fail
string(any:null) -> string


substr -- get substring             call xsubst     may fail
substr(variable:null,integer:0,integer:0) -> variable


tab -- tab through &subject         call xtab       generator
tab(integer:0) -> variable


top -- get stack top                call xtops      may fail
top(stack) -> variable
```

```
trim -- trim string                     call xtrim
trim(string:"",cset:" ") -> string


type -- datatype of argument            call xtype
type(any:null) -> string


upto -- break to character set          call xupto       generator
upto(cset:"",string:&subject,
        integer:&pos or 1,integer:0) -> integer


write -- write line                     call xwrite(n)
write(any list) -> string               (n = number of arguments)


writes -- write string                  call xswrit(n)
writes(any list) -> string              (n = number of arguments)


zz0, ..., zz9 -- system defined         call zz?(l,n)    generator
zz?(any list) -> variable               (n = number of arguments,
                                         l = reactivation label)
```

C.3 Keywords

```
&ascii -> string                        call xkeywd(KASCII)
&clock -> string                        call xkeywd(KCLOCK)
&cset -> cset                           call xkeywd(KCSET)
&date -> string                         call xkeywd(KDATE)
&input -> file                          call xkeywd(KINPUT)
&lcase -> string                        call xkeywd(KLCASE)
&level -> integer                       call xkeywd(KLEVEL)
&null -> null                           call xkeywd(KNULL)
&output -> file                         call xkeywd(KOUTPUT)
&pos -> variable:integer                call xkeywd(KPOS)
&random -> variable:integer             call xkeywd(KRANDOM)
&subject -> variable:string             call xkeywd(KSUBJECT)
&time -> integer                        call xkeywd(KTIME)
&trace -> variable:integer              call xkeywd(KTRACE)
&ucase -> string                        call xkeywd(KUCASE)
```

Appendix E. Pictorial Description of Icon Data Objects

The following figures depict the representation of Icon data objects. The symbols appearing in the figures correspond to the names used in the implementation. For types whose exact representation is machine dependent (such as csets), the DEC-10 representation is shown.

Integers (DINTG = 1)

```
+----------------+      +----------------+
|             ---+----->|          value |
+----------------+      +----------------+
                       in the integer region
```

Strings (DSTRG = 2)

```
                        +----------------+
                        |                |
                        |     actual     |
                        |   characters   |
                        |                |
                         in the string region
                                 .
                                 .
                                 .
+----------------+      +----------------+
|             ---+----->|      SLEN      | length
+----------------+      +----------------+
                        |      SLOC      | character offset
                        +----------------+ into string region
                       in the qualifier region
```

Reals (DREAL = 3)

```
+----------------+      +----------------+
|             ---+----->|      DREAL     |
+----------------+      +----------------+
                        |      RVAL      | real number
                        +                + (occupies 2 cells)
                        |                |
                        +----------------+
                              in the heap
```

Character Sets (DCSET = 4)

```
+----------------+      +----------------+
|             ---+----->|     DCSET      |
+----------------+      +----------------+
                        |     CBITS      | beginning of cset,
                        +                + 1 bit per character
                        |                | (occupies 9 cells)
                        +                +
                        |                |
                        +                +
                        |                |
                        +                +
                        |                |
                        +                +
                        |                |
                        +                +
                        |                |
                        +                +
                        |                |
                        +----------------+
                           in the heap
```

Table Elements (DTENT = 5, see DTABL)

```
+----------------+      +----------------+
|             ---+----->|     DTABL      |
+----------------+      +----------------+
in a table (DTABL)      |     BBREF      | back reference
                        +----------------+
                        |     EREF       | reference
                        +----------------+
                        |     EVAL       | value
                        +----------------+
                        |     ENXT       | next DTENT on hash chain
                        +----------------+
                        |     ETBL       | pointer to table (DTABL)
                        +----------------+
                           in the heap
```

Tables (DTABL = 6)

```
+----------------+      +----------------+
|             ---+----->|     DTABL      |
+----------------+      +----------------+
                        |     BBREF      | back reference
                        +----------------+
                        |     TSIZE      | size of this block
                        +----------------+
                        |     TREFT      | type of reference field
                        +----------------+
                        |     TVALT      | type of value field
                        +----------------+
                        |     TNMAX      | maximum size of table
                        +----------------+
                        |     TNSIZ      | current size of table
                        +----------------+
                        |     TBUCK      | hash bucket[1]
                        +----------------+
                        |                | hash bucket[2]
                        +----------------+
                        |             ---+---> table element block
                        +----------------+        (DTENT)
                        |                |
                        .                .
                        .                .
                        .                .
                        |                |
                        +----------------+
                        |                | hash bucket[n]
                        +----------------+
                           in the heap
```

Lists (1-origined, non-expandable; DLIST = 7)

```
+----------------+      +----------------+
|             ---+----->|     DLIST      |
+----------------+      +----------------+
                        |     BBREF      | back reference
                        +----------------+
                        |     LSIZE      | size of this block
                        +----------------+
                        |     LTYPE      | type of list elements
                        +----------------+
                        |     LELMT      | beginning of elements
                        |                |
                        .                .
                        .                .
                        .                .
                        |                |
                        +----------------+
                           in the heap
```

Lists (arbitrary origin, expandable; DARRY = 8)

```
+----------------+      +----------------+
|             ---+----->|      DARRY     |
+----------------+      +----------------+
                        |      BBREF     | back reference
                        +----------------+
                        |      ASIZE     | size of this block
                        +----------------+
                        |      ATYPE     | type of list elements
                        +----------------+
                        |      AINIT     | initial value
                        +----------------+
                        |      AOPEN     | YES if list is opened
                        +----------------+
                        |      ALBND     | lower bound
                        +----------------+
                        |      AUBND     | upper bound
                        +----------------+
                        |      AELMT     | beginning of elements
                        |                |
                        .                .
                        .                .
                        .                .
                        |                |
                        +----------------+
                           in the heap
```

File (DFILE = 9)

```
+----------------+      +----------------+
|             ---+----->|      DFILE     |
+----------------+      +----------------+
                        |      BBREF     | back reference
                        +----------------+
                        |      FINAM     | internal (integer) name
                        +----------------+
                        |      FSTAT     | status (see below)
                        +----------------+
                        |      FNAME     | string name
                        +----------------+
```

Status Codes (contents of FSTAT):

FCLOS	-1	file is closed
FREAD	0	file is opened for reading
FWRIT	1	file is opened for writing
FRDWR	2	file is opened for reading and writing

Utility block (DUTIL = 10, not a source-language type)

```
+----------------+      +----------------+
|             ---+----->|     DUTIL      |
+----------------+      +----------------+
                        |     USIZE      | size of this block
                        +----------------+
                        |     UDATA      | beginning of non-tended data
                        |                |
                        .                .
                        .                .
                        .                .
                        |                |
                        +----------------+
                            in the heap
```

Procedures (DPROC = 11)

```
+----------------+      +----------------+
|             ---+----->|     DPROC      |
+----------------+      +----------------+
                        |     BBREF      | back reference
                        +----------------+
                        |     PSIZE      | size of this block
                        +----------------+
                        |     PENTRY     | entry point (a label)
                        +----------------+
                        |     PSMAX      | maximum stack size
                        +----------------+
                        |    PPARAMS     | number of parameters
                        +----------------+
                        |    PLOCALS     | number of locals
                        +----------------+
                        |    PSTATIC     | number of static locals
                        +----------------+
                        |     PNAME      | printable name of procedure
                        +----------------+
                        |    PIDENTS     | printable  names of locals
                        |                |
                        .                .
                        .                .
                        .                .
                        |                |
                        +----------------+
                            in the heap
```

Universal Null (DNULL = 12)

```
+----------------+
|              0 +
+----------------+
```

Stack (DSTAK = 13)

```
+----------------+    +----------------+
|             ---+----->|     DSTAK      |
+----------------+    +----------------+
                      |     BBREF      | back reference
                      +----------------+
                      |     SSIZE      | size of this block
                      +----------------+
                      |     STYPE      | type of stack elements
                      +----------------+
                      |     SNMAX      | maximum size of stack
                      +----------------+
                      |      SSP       | stack pointer (an offset)
                      +----------------+
                      |     SELMT      | beginning of elements
                      |                |
                      .                .
                      .                .
                      .                .
                      |                |
                      +----------------+
                         in the heap
```

Records (DRECD = 14)

```
+----------------+    +----------------+
|             ---+----->|     DRECD      |
+----------------+    +----------------+
                      |     BBREF      | back reference
                      +----------------+
                      |     DSIZE      | size of this block
                      +----------------+
                      |     DNAME      | record name as literal index
                      +----------------+
                      |     DTYPE      | type number
                      +----------------+
                      |     DFLDS      | beginning of fields
                      |                |
                      .                .
                      .                .
                      .                .
                      |                |
                      +----------------+
                         in the heap
```

Tended block (DBLOK = 15, not a source-language type)

```
+----------------+      +----------------+
|             ---+----->|     DBLOK      |
+----------------+      +----------------+
                        |     BBREF      | back reference
                        +----------------+
                        |     BSIZE      | size of this block
                        +----------------+
                        |     BDATA      | beginning of tended data
                        |                |
                        .                .
                        .                .
                        .                .
                        |                |
                        +----------------+
                           in the heap
```