A Tour Through the C Implementation of Icon; Version 5.10*

Ralph E. Griswold

William H. Mitchell

TR 85-19

ABSTRACT

This report documents the C implementation of Version 5.10 of the Icon programming language. This report concentrates on the major parts of the system — the translator, the linker, and the run-time system.

August 31, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

A Tour Through the C Implementation of Icon; Version 5.10

Introduction

This report describes the C implementation of Version 5.10 of the Icon programming language [1]. Most of the system is coded in C [2] and is designed to be run under UNIX*. In addition to the C portion of the system, there is some assembly language code. To date, the C implementation has been adapted to the AT&T 3B20, PDP-11†, Ridge 32, Sun Workstation, and VAX-11. This implementation is intended to be portable to other computers running under UNIX, but portability is not a primary design goal. Reference 3 describes the process of transporting this implementation and contains detailed descriptions of the assembly language routines for the VAX implementation.

The implementation of the Icon system consists of three parts: a translator, a linker, and a run-time system. The translator converts an Icon source program into an intermediate form, called *ucode*. The linker combines separately translated ucode files, binds inter-procedure references, and produces interpretable binary output, called *icode*. The run-time system loads and executes an icode file.

The reference language for this report is Version 5.10 of Icon [4]. This report is intended to be used in conjunction with the source listings for Version 5.10, although a general overview of the system can be obtained from this document alone.

The Icon hierarchy is rooted in the directory v5. Source code is in the subdirectory v5/src. This directory in turn has a number of subdirectories, each of which contain a component of the implementation. In the sections that follow, the names of directories, such as tran, are subordinate to src.

1. The Translator

Code for the Icon translator is in the directory tran. The Icon translator is written entirely in C. The translator builds a parse tree for each Icon procedure, then traverses the tree to generate code. Three of the source files in the translator contain only data initialization and are automatically generated from specification files. In addition, the LALR(1) parser is automatically generated by the *Yacc* parser generator [5].

The ucode output from the translator consists of two files. One file, with the suffix .u1, contains intermediate code corresponding to the procedures in the program. The second file, with the suffix .u2, contains global symbol table information.

The following sections discuss the four parts of the translator: the lexical analyzer, the parser, the code generator, and the symbol table manager.

1.1 The Lexical Analyzer

The lexical analyzer reads the Icon source program, breaks it into tokens, and delivers the tokens to the parser as requested. A token is the basic syntactic unit of the Icon language; it may be an identifier, a literal, a reserved word, or an operator.

The lexical analyzer consists of four source files: lex.c, char.c, optab.c, and toktab.c. The latter two of these files contain operator and token tables, respectively, and are automatically generated from operator and token specification files, described below. The file char.c contains character mapping tables and the file lex.c contains the lexical analyzer itself.

The parser requests a token by calling yylex, which finds the next token in the source program and determines its token type and value. The parser bases its actions on the token type: if the token is an operator or reserved word, the token type specifically identifies the operator or reserved word; otherwise, the token type

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

†PDP and VAX are trademarks of Digital Equipment Corporation.

indicates one of the six "primitive" types: identifier, integer literal, real literal, string literal, cset literal, or end-of-file. The token value is a leaf node of the parse tree, which, for the primitive types, contains the source program representation of the token. The token value node also contains the source-program line and column numbers where the token starts. A pointer to this node is placed in the global variable yychar, and yylex returns the token type.

The lexical analyzer finds the next token by skipping white space, including comments. The first character of the new token indicates which of the classes it belongs to. A letter or underscore begins an identifier or reserved word, a digit begins an integer or real literal, a double quote begins a string literal, a single quote begins a cset literal, and any other character is assumed to begin an operator. An identifier or reserved word is completed by gathering all subsequent letters, digits, and underscores. The reserved word table is consulted to determine if the token is an identifier or a reserved word. Numeric literals are recognized by a finite-state automaton, which distinguishes real from integer literals by the presence of a decimal point or the letter "e". A quoted literal is completed by reading until the opening delimiter is repeated, converting escapes in the process and continuing to new lines as necessary. A table-driven finite-state automaton, described below, recognizes operators.

An important task of the lexical analyzer is semicolon insertion. The grammar requires that semicolons separate expressions in a compound expression or procedure body, so they must be inserted into the token stream where they are omitted in the source program. This process is table driven. Associated with each token type are two flags, *BEGINNER* and *ENDER*. The *BEGINNER* flag is true if a token may legally begin an expression (i.e., if it may follow a semicolon). Similarly, the *ENDER* flag is true if a token may legally end an expression (i.e., if it may precede a semicolon). When a newline appears between two tokens, the *ENDER* flag of the first is true, and the *BEGINNER* flag of the second is true, then a semicolon is inserted between the two tokens.

The file toktab.c contains the initialization for the token table. The table is divided into three sections: primitive types, reserved words, and operators. The primitive types are fixed in the first six slots in the table, and must not be changed, since they are referenced directly from the code. The reserved words follow and must be in alphabetical order. The operators follow in no special order. The last entry merely marks the end of the table.

Also in toktab.c is an index to reserved words. To speed up the search for reserved words, this table hashes the search using the first letter as the hash value. The reserved words that begin with that letter then are examined linearly.

The operator table, in optab.c, describes a finite-state automaton that recognizes each operator in the language. Each state is represented by an array of structures. Each structure in the array corresponds to a transition on the input symbol. The structure contains three fields: an input symbol, an action, and a value used by the action. The recognizer starts in state 0; the current input symbol is the first character of the operator. In a given state with a given input symbol, the recognizer searches the array associated with the current state for an entry that matches the current input symbol. Failing a match, the last entry of the array, with the input symbol field of 0, is used. The recognizer then performs one of the following actions, depending on the value of the action field:

- Goes to the new state indicated by the value field and gets the next input character.
- Issues an error.
- Returns the value field as a pointer to the token table entry for the operator.
- Returns the value field, but pushes the current input character back onto the input.

The difference between the last two actions is that some operators are recognized immediately (e.g., ";"), while others are not recognized until the character following the operator is read (e.g., "=").

The token table and operator table are constructed by the Icon program mktoktab.icn. This program reads the specification file tokens and builds the file toktab.c. The file tokens contains a list of all the tokens, their token types (given as defined constants), and any associated flags. This list is divided into the three sections detailed above. The program then reads the specification file optab and builds the file optab.c. The former is a skeleton for the operator table; it contains the state tables, but the program fills in the pointers to the token table entries.

1.2 The Parser

The parser, in the file parse.c, is automatically generated by *Yacc*. The grammar and semantic actions are contained in the file icon.g. From these specifications, *Yacc* generates parser tables for an LALR(1) parser.

In addition to the grammar, icon.g contains a list of all the token types in the language and declarations necessary to the actions. *Yacc* assigns an integer value to each token type, and generates define statements, which are written to the file token.h. These defined constants are the token types returned by the lexical analyzer.

The grammar is context-free, with actions associated with most of the rules. An action is invoked when the corresponding rule is reduced. The actions perform two duties: maintaining the symbol tables and constructing the parse tree. The parse tree is built from the bottom up — the leaves are supplied by the lexical analyzer and the actions build trees from the leaves and from smaller trees with each reduction.

The parser requests tokens from the lexical analyzer, building a parse tree until it reduces a procedure. At this point, it passes the root of the parse tree to the code generator. Once the ucode has been generated, the parse tree is discarded, and a new tree is begun for the next procedure.

Record and global declarations affect only the symbol table and do not generate parse trees. Files named in link directives produce link instructions in the ucode output.

A complete parse tree is rooted at a proc node, which identifies the procedure and points to the subtrees for the initial clause (if any) and the body of the procedure. Each node in the parse tree represents a source program construction or some implicit semantic action. A node can contain up to six fields, the first of which is the node type. The second and third fields are always line and column numbers that are used for error messages and tracing. Any additional fields contain information about the construction, and possibly pointers to subtrees. Appendix A contains a description of all the node types.

The grammar, shown in Appendix B, has several ambiguities. The well-known "dangling else" problem exists not only in the if-then-else expression, but also in the while-do, until-do, every-do, and to-by expressions. In each of these expressions, the last clause is optional, so that when the parser sees an else, for example, it does not know whether to shift the token (associating it with the most recent if), or to reduce the preceding if-then expression (leaving the else "dangling"). The latter choice is obviously incorrect, since the else would never be shifted, and *Yacc* correctly resolves such conflicts in favor of the shift. Thus, each else is paired with the most recent unpaired if. All the control structures (except case) have an additional ambiguity: they do not have a closed syntax, yet they may appear in an expression at the highest precedence level. For example, the expression

```
x := y + if a = b then z else -z * 3
```

could parse in either of two ways:

```
x := y + (if a = b then z else (-z * 3))
x := y + (if a = b then z else -z) * 3
```

This problem, too, is resolved in favor of the shift, such that the first parse is always used. Thus, in the absence of parentheses, the entire expression to the right of a control structure is part of the control structure.

Little attention has been paid to error recovery. A few error productions have been placed in the grammar to enable *Yacc* to recover from syntax errors; the technique for doing so is described by Aho and Johnson [6]. The parser is slightly modified by the editor script pscript so that the parser state is passed to the routine yyerror. This routine prints an error message from the file synerr.h that is associated with the current parser state. This error table currently is constructed by hand from the y.output file obtained by running *Yacc* with the -v option.

1.3 The Code Generator

The parser calls the code generator upon recognition of each Icon procedure, giving it the root of the parse tree. The code generator traverses the parse tree recursively, emitting ucode. Appendix C contains a description of ucode.

The file code.c contains both the tree node allocation and the code generation routines. The header *file* code.h contains macros and definitions used by the code generator, while tree.h defines the tree nodes and the macros that allocate them. Thus, the macros in tree.h provide the interface between the parser and the code generator.

The tree traversal routine, traverse, is a recursive procedure with one argument, a pointer to the root of a tree or subtree for which code is to be generated. The routine examines the type field of the root and, through a switch statement, generates a sequence of ucode instructions as determined by the type. If the node has subtrees, traverse calls itself recursively at the appropriate point to generate code for the subtree. For example, the code generated for a binary operator first generates code for its two subexpressions, then emits the code that calls the appropriate run-time library routine.

The returned value of the traversal routine is used for counting elements of expression lists. If the root of the tree being traversed is an elist (expression list) node, traverse returns the sum of the returned values of its two subtrees. Otherwise, it returns 1. This count is used when generating code for procedure calls and lists with explicit elements, which need to know the number of arguments to be pushed onto the stack.

When generating code for loops, the code generator needs to save three pieces of information for each nested loop: the *break label*, the *next label*, and the expression nesting level. This information is kept on the *loop stack*. The break label is a label placed just past the end of the loop; it is the place where control is passed when the loop is finished. The next label is placed near the end of the loop, at a point where the next iteration of the loop can be started. The code for break and next expressions branches to these labels, but in either case, any incomplete expression frames (see Section 3.2) within the loop must first be popped from the stack. The expression nesting level counts the number of currently active expression frames within the current loop; an unmark instruction is generated for that many expression frames (less one for a next expression).

The possibility of nested case expressions requires that certain information be kept on a *case stack*. For each case expression, the code generator allocates a label for the end of the expression and pushes it onto the case stack. When a default clause is encountered, its subtree is placed on the top of the case stack to delay code generation for it until the end of the case expression.

1.4 The Symbol Table Manager

The symbol table manager consists of the symbol table data structures and routines that operate upon these data structures. The source code for the symbol table manager is contained in two files. The file keyword.c contains only the keyword name table and is automatically constructed from a keyword specification file discussed below. The remainder of the symbol table manager is located in the file sym.c.

The symbol table manager operates with two logical data structures, the symbol table proper and the string space. When the lexical analyzer identifies a token as either an identifier or a literal, the lexical analyzer requests the symbol table manager to enter the token into the string space. The symbol table manager returns a pointer into the string space for that string. The lexical analyzer then places this pointer in the token value node. To help keep the size of the string space small, all entries are hashed, and only one copy of any string is kept. This has the added benefit that two strings can be compared by checking only the pointers into the string space.

The parser determines the context of the token and requests the symbol table manager to enter the token into the symbol table proper. It is the responsibility of the symbol table manager to verify that the use of the token is consistent with prior use. Appropriate diagnostics are issued if the use is inconsistent.

The symbol table proper is physically divided into three separate structures: the *global*, *local*, and *literal* tables. Each of these tables is hashed, using the pointer into the string space as the key. Since this pointer is an offset into the string space, hashing is simply and effectively performed by taking the rightmost *n* bits of the offset (where 2^n is the size of the hash vector for the table).

The global table contains identifiers that have been declared as globals, procedures, or records. The local table holds all identifiers declared as locals, formal parameters for procedure declarations, field names for record declarations, and all other identifiers referenced in the procedure (including those declared as global elsewhere). The literal table contains entries for literal strings and csets, integers, and floating-point constants.

Both the local and literal tables are associated with the current procedure being parsed and are written to the .u1 file when the procedure has been successfully parsed. If a record declaration has been parsed, then the local table, containing only the field name identifiers, is written to the .u2 file. After all procedure, record, and global declarations in a Icon source file have been parsed, the global table is written into the global declarations file.

An entry into any of the three symbol table sections is a structure with three fields: a link, a name, and a flag. The link field holds the pointer to the next entry in the same hash bucket. The name is the pointer to the identifier or literal name in the string space. The flag field contains the type (*formal parameter*, *static local*, *procedure name*, etc.) of the entry. Global table entries have a fourth field, an integer providing the number of formal parameters for a procedure declaration, or the number of fields in a record declaration.

Lookup in the local and global tables is merely the process of following a hash chain until an entry of the same name is found or until the hash chain is exhausted. If a previous entry is found, the flags of the existing and new entries are compared, and diagnostics are printed if the use of the new entry conflicts with the previous usage. The new entry is ignored whenever such an inconsistency is found.

The literal table uses the same lookup procedure, except the search down the hash chain stops when an entry is found with the same textual form and flag fields. Thus the string literal "123" and the integer literal 123 have separate entries in the literal table, even though they have the same string representations. A consequence of this technique is that the integer literals 123 and 0123 have separate entries in the literal table, even though they have the same numeric value. Since most programmers use a reasonably consistent style when expressing literals, this technique usually does not produce many duplicate constants.

A final task of the symbol table manager is the identification of keyword names. (Note that keywords are of the form &*name*.) The symbol table manager maintains a list of the legal keyword names and, upon request, returns a numeric identification for a keyword name to the parser. An automatic procedure exists for creating the keyword name table: the Icon program mkkeytab.icn reads the specification file keywords and produces the keyword name table in keyword.c. The file keywords is simply a list of the keyword names and a numeric identification for each. Since the number of keyword names is small, and only a few references to keywords are typical in an Icon program, lookup in the keyword name table is done using a linear search.

The sizes of the respective portions of the symbol table may be altered with command line arguments to the Icon translator.

2. The Linker

Code for the Icon linker is in the directory link. The Icon linker is written entirely in C. The linker performs three tasks: combining the global symbol tables from one or more runs of the translator, resolving undeclared identifiers, and translating ucode to icode. The resulting combined global symbol table is used for determining the scope of undeclared identifiers during the second task. The second and third tasks are done during a single pass over each ucode file. A single icode file is produced.

The symbol table module, in the file lsym.c, is similar to the symbol table module of the translator, except that there is an additional table for storing field names of records. The input module, in the file llex.c, recognizes the instructions in ucode files. The global symbol tables are merged by the routine globals in glob.c, and the icode file is produced by the routines in lcode.c. Of the remaining source files, ilink.c and lmem.c contain the main program, miscellaneous support routines, and memory initialization. The files builtin.c and opcode.c contain table initializations for the list of built-in functions and the ucode operations, respectively.

The first phase of the linker reads global symbol information from each .u2 file and enters all the global symbols into one combined table. The format of a global symbol table file is described in Appendix C. This phase also builds the record/field table that cross-references records and field names, and sets the trace flag for execution-time tracing if any of the files being linked were translated with the -t option. As records are entered into the global symbol table and the record/field table, they are numbered, starting from 1. These record numbers are used to index the record/field table at run-time when referencing a field.

When the linker encounters a link instruction, the named file is added to the end of a linked list of files to be linked. The list initially consists of the files named as arguments. Names are not added to the list if they are already on it.

The second phase reads each .u1 file in sequence, emitting icode as each procedure is encountered. Appendix C describes the ucode instructions. The .u1 files contain a prologue for each procedure, beginning with a proc opcode, followed by a series of loc opcodes describing the local symbol table, a series of con opcodes describing the constant table, and a declend opcode terminating the prologue. The local symbol table contains not only local symbols, but all identifiers referenced in the procedure — global, local, or undeclared. When an undeclared identifier is entered into the local symbol table, its scope is resolved by the following steps:

- If the identifier has been entered in the global symbol table, it is entered into the local symbol table as a global identifier.
- If the identifier matches the name of a built-in function, it is entered into the local symbol table as such.
- Otherwise it is entered as a local identifier and a warning is issued if the linker is run with the -u option.

The constant table contains an entry for each literal used in the procedure.

The linker outputs icode in several regions. The first region contains icode instructions and procedure blocks, as well as blocks for cset and real literals. The next region contains the record/field table and procedure blocks for record constructors. The next four regions contain the values of the global identifiers, the names of the global identifiers, the values of the static identifiers, and strings.

An icode instruction consists of an opcode and, in some cases, operands. The sizes of opcodes and operands depend on the machine architecture and the implementor's judgement. On the VAX-11, opcodes are one byte long and operands are four bytes long. Most instructions correspond exactly to instructions in the ucode that is output by the translator. The opcode values are those used internally by the linker (defined in the file link/opcode.h).

Fields are provided in the global symbol and literal tables for associating an icode location with each entry. As the prologue is being read, each cset, real, or long-integer literal entered into the literal table is output immediately and its location is stored in the literal table. Thus, the locations of all literals are known before their reference.

The same is true of references to procedures, since these references only occur in the initialization for global identifiers, which is not output until all procedures have been output. When the prologue for a procedure has been completely processed, the procedure block is output, and its location is noted in the global symbol table.

References to program labels require backpatching, since there often are forward references. Because program label references are always local to the current procedure, the linker buffers the output code for a procedure. A table of values for all program labels is initialized to zero at the beginning of each procedure. When a label is referenced and its table entry is zero, the location of the reference is negated and stored in the table entry and a zero is output for the operand. If a label's table entry is negative, the location of the reference is negated and stored in the table entry as before, but the previous value of the table entry is output for the operand. This forms a linked list of references to the as-yet-undefined label. When a label is defined, each reference on the linked list is replaced with the correct value of the label.

References to global and static identifiers are determined at run-time. The global and static instructions have an integer operand referring to the identifier by position in the global or static identifier array. When one of these instructions is interpreted, the actual address is calculated from the position and the known address of the global or static identifier array. References to functions are also resolved at run-time. Each function is assigned an integer index (its position in the table of functions in builtin.c). When the global identifier initialization for a function is output, the negated index is output instead of an address. The interpreter fills in the correct address during program initialization.

Once the prologue has been processed, a procedure block (see Section 3.1) is emitted. Opcodes following the prologue represent execution-time operations, and cause code to be emitted.

The record/field table is a two-dimensional matrix, first indexed by a field number assigned to each identifier that is used as a field name, next by a record number assigned to each record type. The value at the selected position in the table is the index of the field in a record of the given type, or -1 if the given record type does not contain the given field.

The initial value for global and static identifiers is the null value unless the global identifier is a procedure, function, or record constructor, in which case the initial value is a descriptor of type procedure that points to the appropriate procedure block. The values output use the data representations described in Section 3.1.

The names of global and static identifiers are output as *qualifiers* (see Section 3.1) and are used by the function display. All qualifiers contained in the generated procedure blocks and global and static names point into a section of the icode file that contains character data. String literals also point to this section of the icode.

3. The Run-Time System

The run-time system consists of an interpretive loop and a collection of routines that collectively provide support for the execution of an Icon program.

Three directories contain routines relating directly to source-language operations: src/fncs, src/ops, and src/lib. The first two directories contain one routine per function or operator, respectively. The lib directory contains routines relating to Icon control structures. A fourth directory, rt, contains routines for performing common operations needed by many routines in the other three directories. In particular, rt contains routines that handle storage allocation and reclamation, type conversion, data comparison, integer arithmetic with overflow checking, generator suspension, and tracing. The directory iconx contains initialization and the interpreter. Code in these directories is largely machine-independent. Corresponding machine-dependent code is contained in src/sys. The object files in these directories are loaded together to form the run-time system, which is named iconx.

Most of the run-time system is coded in C, but some of the routines are coded in assembly language. The interpretive loop and startup routines are written in assembly language, as is integer arithmetic with overflow checking (C does not provide this), as well as routines concerned with stack management. All assembly-language code is contained in src/sys.

Before iconx begins executing an Icon program, it reads in the icode file generated by the linker. The first eight words of this file contain header information indicating the total size of the rest of the file, the initial value of &trace, and the relative offsets from the beginning of the file to the various sections. These offsets are converted to actual addresses by adding the base address of the icode region. Several pointers in the icode must also be relocated. The interpreter sweeps through the global identifiers, looking for procedures, functions, and record constructors. For each function, it supplies the address of the appropriate procedure block. For each procedure, it relocates pointers from its procedure block to the procedure entry point, as well as to strings representing the procedure and local identifier names in the identifier table. For each record constructor, it supplies the address of mkrec, the routine that constructs new records, as the entry point in the procedure block.

The interpreter then begins execution by invoking the value of the first global identifier, which corresponds to the procedure main. If there is no main procedure, the first global identifier has the null value and a run-time error is reported. The routine invoke sets the *interpreter program counter* (*ipc*) to the entry point, and branches to interp, which is contained in src/sys/interp.s.

The routine interp is the main interpreter loop. It fetches the next opcode, and branches to the appropriate processing routine through a jump table.

3.1 Data Representations

Icon has two elementary forms of data objects — values and variables. Values often can be converted from one data type to another. When this is done automatically, it is called *coercion*. The process of obtaining the value referred to by a variable is called *dereferencing*.

Every data object is represented by a two-word *descriptor*, which may, depending on the type of the object, contain a value or refer to some other area of memory for the actual value. The first word of the descriptor, referred to as the *d-word*, contains flags, a type code, or other similar information. The second word, referred to as the *v-word*, either contains the value or a pointer to it. There are six descriptor formats, shown in Appendix D: *qualifier*, *null*, *integer*, *pointer*, *variable*, and *trapped variable*. These formats are distinguished from one another by the four high-order bits of the d-word. In integer, pointer, and trapped variable descriptors, the least-significant six bits of the d-word identify the type of object represented, while the most significant bits in the d-word are flags that classify the object (for example, whether the second word contains a pointer — historically, a "floating address" [7]).

A qualifier represents a string, and contains the length of the string and a pointer to the first character of the string. The null descriptor represents the null value. An integer descriptor represents an integer small enough to fit in the second word of the descriptor. This includes all integers on computers whose C *ints* are the same size as C *longs* (such as the VAX-11). All data types other than integer, string, and null are represented by pointer descriptors. A pointer descriptor contains a pointer to a block of appropriate format for a value of the given type. On computers whose C *longs* are longer than C *ints* (such as the PDP-11), an integer that requires more bits than there are in an *int* is contained in a *long integer* block. The block formats for each data type are shown in Appendix D.

The v-word of a variable descriptor either points to a fixed location (in the case of global and static identifiers), a location on the stack (in the case of local identifiers), or it points to a descriptor in a block, which is the case for list elements, for example. A variable that points to a descriptor in a block also contains an offset that indicates the distance (in words) from the beginning of the block to the referenced descriptor. This offset is used during the marking phase of garbage collection, which is discussed in Section 3.3.

A trapped variable [8] descriptor represents a variable for which special action is necessary upon dereferencing or assignment. Such variables include substrings, non-existent elements of tables, and certain keywords. Each type of trapped variable is distinguished by a type code in in its d-word.

Substring trapped variables, created by a section or subscripting operation, contain a pointer to a block that contains a variable descriptor identifying the value from which the substring was taken, an integer indicating the beginning position of the substring, and an integer showing the length of the substring. With this information, assignment to a substring of a variable can modify the contents of the variable properly. Substrings of non-variables do not produce substring trapped variables, since assignment to such substrings is meaningless and illegal; instead, forming the substring of a non-variable produces a qualifier.

Table element trapped variables, formed by referencing elements of tables, similarly contain a pointer to a block that contains enough information for assignment to add the element to the referenced table or to supply the default table value.

The keywords &pos, &random, &subject, and &trace are handled via *keyword trapped variables*. A keyword trapped variable points to a block that contains the value of the keyword as well as a pointer to a function that performs assignment to the keyword. A function is needed, since in the case of these keywords, assigned values must be checked for validity and possibly coerced to the expected type.

Strings formed during program execution are placed in the *allocated string region*; qualifiers for these strings point into this region. Substrings of existing strings are not allocated again; instead, a qualifier is formed that points into the existing string. When storage is exhausted in the allocated string region, the garbage collector (see Section 3.3) is invoked to reclaim unused space and compact the region; if enough space cannot be reclaimed, the region is expanded if possible.

Data blocks formed during program execution are placed in the *allocated block region.* Data blocks have a rigid format dictated by the garbage collection algorithm. The first word of the block, referred to as its *title*, always contains a type code that identifies the structure of the rest of the block. Descriptors follow all non-descriptor information in the block. If the size of the block is not determined by its type, the size (in bytes) is contained in the second word of the block.

When storage is exhausted in the allocated block region, the garbage collector is invoked to reclaim unused space and compact the region; if enough space cannot be reclaimed, the region is expanded if possible.

Co-expression stack blocks are allocated in a separate region that lies below the allocated string and block regions. Co-expression stacks are allocated and collected using a free-list strategy.

3.2 Stack Organization

The stack is the focus of activity during the execution of an Icon program. All operators, functions, and procedures find their arguments at the top of the stack, and replace the arguments with the result of their computation. The values of arguments and local identifiers for Icon procedures are also kept on the stack. The arguments, local identifiers, and temporaries on the stack for an active Icon procedure are collectively called a procedure frame. This is one of several kinds of stack frames discussed in this section. See [3] for a detailed discussion of stack frames. Each co-expression also has a stack. For uniformity, the main stack is treated as the stack for the co-expression &main.

On the PDP-11 and VAX-11 stacks start in high memory and grow downward. On these computers, a push causes the stack pointer to decrease and a pop causes the stack pointer to increase. Thus "above" and "below" refer, respectively, to "newer" and "older" information on the stack. Note that the top of the stack is the low word. The description of relative stack locations that follows is based on this kind of architecture and nomenclature.

Before an Icon procedure calls another Icon procedure, the caller pushes the procedure to be called (a descriptor, since procedures are data objects in Icon) onto the stack. The caller then pushes each argument (also a descriptor) onto the stack, leftmost argument first. The caller then pushes one word onto the stack indicating the number of arguments supplied, which may be different from the number of arguments expected. The procedure is then invoked. Invocation checks that the first descriptor pushed above actually does represent an integer, procedure, or a variable whose value is an integer or a procedure. An integer indicates the selection of one of the arguments resulting from mutual evaluation. A procedure, on the other hand, points to a procedure block, which contains various information about the called procedure, including the number of arguments expected, the number of local variables used, and the procedure's entry point address. Next, the number of arguments supplied to the procedure is adjusted to match the number expected, deleting excess arguments or supplying the null value for missing ones. This adjustment is performed by moving the portion of the stack below the arguments up or down, as appropriate. It then dereferences the arguments. A *procedure marker* is then pushed onto the stack, and the *procedure frame pointer* is set to point to the new procedure marker. The procedure marker contains, among other things, the return address in the calling procedure and the previous value of the procedure frame pointer. Next, the null value is pushed onto the stack as the initial value for each local identifier. Control then is transfered to the procedure's entry point, and execution of the Icon program resumes in the new procedure.

When an Icon procedure is ready to return to its caller, it pushes its return value (a descriptor) on the stack. It then transfers control to pret, which moves the return value to the location occupied by the descriptor that represented the called procedure. That is, the return value is stored in place of the first descriptor that was pushed at the beginning of the calling sequence described above. The return sequence then restores the state of the previous procedure from the current procedure marker (the procedure marker that the procedure frame pointer currently points to). This includes restoring the previous value of the procedure frame pointer, retrieving the return address, and popping the returning procedure's local variables, procedure marker, and arguments. Thus, when the calling procedure regains control, the arguments have been popped and the return value is now at the top of the stack.

Functions and operators are written in C, and therefore obey the C calling sequence. By design, the Icon calling sequence described above is similar to the C calling sequence. When an Icon procedure calls a function, a *boundary* on the stack is introduced, where the stack below the boundary is regimented by Icon standards, and the stack above the boundary contains C information. This boundary is important during garbage collection: The garbage collector must ignore the area of the stack above the boundary, since the structure of this area is unknown, whereas the structure of the area below the boundary is well-defined. In particular, all data below the boundary is contained in descriptors or is defined by the structure of a frame, so that all pointers into the allocated string or block regions may be located during a garbage collection.

Functions and operators are written to "straddle" the boundary. From below, they are designed to resemble Icon procedures; from above, they are C procedures. An Icon procedure calls a function in the same way as it calls another Icon procedure; in fact, functions are procedure-typed data objects just as Icon procedures are. When invoke recognizes that a function is being called, it bypasses the argument adjustment if the field in the procedure block that indicates the number of arguments expected contains -1, which indicates that the function can take an arbitrary number of arguments. It also does not allocate stack space for local variables, since any such variables are C variables and are allocated by the C function itself. C routines have an entry sequence that creates a new procedure frame; since invoke has already done this, such routines are entered by branching to the first instruction past the entry sequence.

For functions that take a fixed number of arguments, these arguments are names arg1, arg2, ... For functions that can take an arbitrary number of arguments, there is a macro Arg(*n*) that accesses the *n*th argument Thus, Arg(1) accesses the first argument (as a descriptor), and Arg(nargs) accesses the last argument. Each function is responsible for supplying defaults for missing arguments. Because of the calling protocol, the returned value is in arg0. Every function places its result there and then returns through normal C conventions. Each function also supplies a procedure block that contains the number of arguments expected (or −1), its entry point, and a qualifier representing its name.

Operators are very similar to functions. The only difference is that operators are called directly (rather than being called through invoke) and must set the boundary themselves.

When an operator or function fails to produce a result, it calls fail. This routine initiates backtracking as described below.

Expressions are evaluated within an *expression frame*. When the evaluation of an expression is complete, whether it has produced a result or failed, the expression frame must be popped from the stack and the result of the expression must be pushed back onto the stack. The expression frame marks the stack height at the point that the expression began to be evaluated, so that the stack may be restored to its original state when the evaluation of the expression is complete. The stack normally would be restored to the original height (that is, the pops would match the pushes) except when an expression fails at some midpoint in its evaluation. The expression frame is also used to limit backtracking: backtracking is restricted to the current expression instance only.

When evaluation of an expression begins, an *expression marker* is pushed on the stack, the *expression frame pointer* is set to point to it, and the *generator frame pointer*, discussed below, is cleared. The marker contains the previous values of the expression and generator frame pointers and a failure label. When an expression produces a result, that result, on the top of the stack, is popped and saved. Then the stack is popped to the expression marker, and the previous values of the two frame pointers are restored. The marker is popped and the result of the expression is pushed back onto the stack, now a part of the previous expression frame. If an expression fails to produce a result, efail pops the stack to the expression marker, restores the previous values of the two frame pointers, and branches to the failure label. In the special case that the failure label is zero, efail is effectively called again to indicate failure in the new expression frame. Thus the failure is propagated from one expression to an enclosing one.

If an expression has any generators, then there is a *generator frame* within the current expression frame for each generator that is inactive (that is, that has produced a value but is not yet exhausted). A generator frame preserves the state of the stack at the point just before the generator (whether it be operator, function, or procedure) suspended (became inactive). If efail is called and there are inactive generators, then instead of exiting the current expression frame, the most recently inactivated generator is reactivated by restoring the stack to the state saved in the most recent generator frame.

A function or operator suspends itself by calling suspend. This routine preserves the state of the stack by duplicating the current expression frame, bounded on one end by the most recent generator frame or by the current expression frame, if there are no inactive generators. and bounded on the other end by the beginning of the argument list of the suspending function or operator. A generator marker is pushed onto the stack, followed by the duplicate expression frame. The routine suspend then causes the suspending function or operator to return to its caller, instead of itself returning.

When reactivated by efail, the stack is restored to the generator marker, which is used to restore the various frame pointers. Then the marker is popped. The stack is then in the same state that it was in when suspend was called. The routine efail then returns to the generator as if the original call to suspend had returned. Thus the following schema is typical of operators and functions that generate a sequence of values.

```
initialize;
while (not exhausted) {
   compute next value;
   store return value;
   suspend()
   }
fail();
```

The effect of resuming an expression containing generators is that suspend actually causes the generator to return. If alternatives are needed, backtracking occurs, and the apparent effect is that suspend has returned. The generator computes the next result, and suspends with it. When the generator is exhausted, it merely fails without suspending, which just passes the failure back to the next most recently inactivated generator, if any.

Just as functions and operators can return normally, suspend, or fail, so can Icon procedures. The mechanics are essentially the same, but the differences in stack layout require different primitive operations. When Icon procedures return normally, the return value is presumed to be at the top of the stack and pret is called. Similarly, Icon procedures call psusp to suspend. Both of these routines also dereference the return result if it is a local variable. The routine pfail causes an Icon procedure to return with no result.

The same three primitives are also needed at the expression level: eret, esusp, and efail. eret is not a library routine, but is generated as in-line code. Both cause an exit from the current expression frame; but eret supplies a result to the enclosing expression, while unmark does not. The routine esusp creates a inactive generator before supplying a result to the enclosing expression; it is used by the alternation control structure. The routine efail simply causes backtracking within the current expression frame. In fact, fail and pfail merely exit their procedure frame before branching to efail.

3.3 Storage Allocation and Reclamation

During program execution, storage is allocated as necessary when data objects are created. The three primitive routines allocate, alcstr, and alcestk allocate storage in the block, string, and co-expression stack regions, respectively. All three routines return pointers to the beginning of newly allocated space. None of the routines is responsible for ensuring that enough space remains in the data regions. Ensuring that enough space remains in the data regions is the responsibility of a *predictive need* strategy described below.

In the allocated block region, allocate(n) returns a pointer to n contiguous bytes of storage. Because a wide variety of objects may reside in the allocated block region, a number of support routines are provided to simplify the storing of various objects. There is a specific routine to allocate a block for each datatype in the block region. Where appropriate, these routines have the actual values to be stored as their arguments. All of the routines call allocate to obtain storage for the object.

In the string region, alcstr(s, l) allocates room for a string of length l and copies the string pointed to by s into this space. Since some routines such as left, right, and center need room in the string space in which to construct a string, a call to alcstr with the defined constant NULL as the first argument results in the allocation of storage without attempting to copy a string.

In the co-expression stack region, alcestk() allocates a new co-expression stack.

Source code for all of the allocation routines is contained in rt/alc.c.

As mentioned earlier, a *predictive need* strategy is employed to ensure that enough room remains for data storage. Simply put, *predictive need* states that it is the responsibility of any routine that calls an allocation routine both to ensure that enough room remains in the proper data region and to maintain the validity of any temporary pointers into the data regions, should a garbage collection be necessary to free storage space.

Since the check for storage space only needs to occur before the allocation takes place, each routine may perform this check at its convenience. This approach permits the minimization of the number of temporary pointers that must be protected during garbage collection.

Routines to ensure space are provided for each of the three storage regions. The routine strreq(n) ensures that at least n bytes of storage remain in the string space, and blkreq(n) performs the same function in the allocated block region. stkreq() ensures that there is a co-expression stack available. If any of these routines finds that there is insufficient storage remaining, it invokes the garbage collector in an attempt to obtain that storage. If that fails, then program execution is aborted with an appropriate diagnostic.

Garbage collection is a process that identifies all valid allocated data. In the string and block regions, valid data is compacted in order to provide a contiguous area of unused storage. In the co-expression stack region, unused co-expressions are returned to a free list. The algorithm used for identifying valid data is based upon the algorithm described by Hanson [7]. Only the more novel features are discussed here.

Whenever a predictive need request discovers that insufficient storage remains in one of the regions, the garbage collector is invoked to reclaim unused space in all regions. This approach is more efficient in situations where all regions are heavily allocated and only slightly less efficient otherwise.

The approach is to sweep through the static data regions and the stack, looking for qualifiers that point into the allocated string region and descriptors that point into the allocated block region. When a qualifier is found, a pointer to that qualifier is saved in a temporary data region at the end of the allocated block region.

If the descriptor is a pointer into the allocated block region, then that block contains valid information. The block is marked as valid, the descriptor is placed on a back chain headed in the block, and the marking process is called recursively on any descriptors within that block. Blocks that are already marked as valid are not processed a second time. To simplify the marking of blocks, all blocks have been designed so that all descriptors within them exist as a contiguous section at the end of the block. Thus, to sweep through the descriptors within a block, the marking algorithm need only know the size of the block and the location of the first descriptor. Information concerning a block's size, as well as the offset for the first descriptor is in the file rt/dblocks.c.

Valid co-expression stacks also may contain qualifiers and pointers to other valid data; such stacks are included in the marking phase.

After the marking phase is completed, the string region is compacted. The algorithm used is described by Hanson [9]. The pointers to the qualifiers are sorted so that the order of all valid strings within the string space is identified. The qualifiers are then processed in order, and modified as the valid strings are compacted. If this compaction does not free enough space within the string region to satisfy the request, the allocated block region must be moved in order to provide more room in the string region. An attempt is also made to provide some additional "breathing room" in the string region to permit future allocation.

The allocated block region cannot be moved until after the valid pointers into it are adjusted and the storage is compacted. The pointer adjustment and compaction phases are two linear passes through the allocated block region. The only difference when the allocated block region is to be moved is that the adjusted pointers point to where that data will be after the region has been moved. If not enough breathing room is freed in the allocated block region, then more space is requested from the operating system. As a last step, if the string region needs more room, the block region is relocated.

A shortcoming of the implementation is the absence of a process for decreasing the size of a data region, should it become too large. It is also possible that insufficient room would be available for storing the pointers to the qualifiers, even though enough storage would become available if the block region were collected separately. In practice, this has not been a problem. The source code for the garbage collector is contained in the files sys/gcollect.s, rt/gc.c, and sys/sweep.c.

3.4 Coding Conventions

The calling conventions for functions and operators have been mentioned earlier. Several other aspects of the run-time system are explained here.

All header files for the run-time system are in the directory h. The file h/rt.h is included by every source file in the run-time system, and contains machine-independent defined constants, run-time data structure declarations, and defined constants and macros for flags, type codes, argument accessing, and bit manipulations. The file sys/params.h contains corresponding machine-dependent information.

During the execution of an Icon program, many type conversions are done on temporary values, where data storage is not required beyond the bounds of the current operation. For this reason, the type conversion routines all operate with pointers passed to them that reference buffers in the calling routine. Any routine calling for type conversion must determine if space is needed in the block or string regions, and perform the allocation. Most of the conversion routines return the type of the result or NULL if the conversion cannot be performed. One exception is cvstr which, in addition to NULL, returns 2 if the object was already a string, and 1 if the object had to be converted to a string. This distinction makes it possible to avoid a large number of predictive-need checks. The second exception is cvnum, which returns either real numbers or integers and makes no attempt to distinguish between short and long integers.

The garbage collector knows about the descriptors arg0, arg1, ... that are the arguments of C routines for functions and operators. These descriptors are "tended" during garbage collection. The garbage collector does not know about other descriptors that may be declared in C routines and does not tend them. All pointers to data in allocated storage regions must be tended when a garbage collection occurs. Since a garbage collection can occur only during a call to strreq, blkreq, or stkreq, or between suspension and reactivation, the only places where C routines need to ensure that all pointers into the allocated block and string regions are tended are just before calls to strreq, blkreq, stkreq, or suspend.

Acknowledgements

This report is a revision of earlier descriptions of the C implementation of Icon [10, 11, 12], which were co-authored by Cary Coutant, Rob McConeghy, and Steve Wampler. Much of the material in this report is taken from the earlier ones.

Many features of the current implementation of Icon are based upon the original Ratfor implementation by Dave Hanson, Tim Korb, and Walt Hansen [13, 14].

References

1. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.
2. Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.
3. Mitchell, William H. *Porting the UNIX Implementation of Icon; Version 5.10.* Technical Report TR 85-20, Department of Computer Science, The University of Arizona, Tucson, Arizona, August 1985.
4. Griswold, Ralph E. and William H. Mitchell. *Version 5.10 of Icon.* Technical Report TR 85-16, Department of Computer Science, The University of Arizona, Tucson, Arizona, August 1985.
5. Johnson, Stephen C. "Yacc: Yet Another Compiler-Compiler" *Unix Programmer's Manual, Seventh Edition.* Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, 1979.
6. Aho, A. V., and S. C. Johnson. "LR Parsing" *Computing Surveys* **6**, 2 (June 1974), 99-124.
7. Hanson, David R. "Storage Management for an Implementation of SNOBOL4", *Software—Practice and Experience* **7**, 2 (March 1977), 179-192.
8. Hanson, David R. "Variable Associations in SNOBOL4", *Software—Practice and Experience* **6**, 2 (April 1976), 245-254.
9. Hanson, David R. *The Manipulation of Variable-Length String Data in Fortran IV.* Technical Report, Department of Computer Science, The University of Arizona, Tucson, Arizona, May 1975.
10. Coutant, Cary A. and Stephen B. Wampler. *A Tour Through the C Implementation of Icon; Version 5.* Technical Report TR 81-11a, Department of Computer Science, The University of Arizona, Tucson, Arizona, December 1981.
11. Griswold, Ralph E., William H. Mitchell, and Stephen B. Wampler. *The C Implementation of Icon; A Tour Through Version 5.* Technical Report TR 83-11a, Department of Computer Science, The University of Arizona, Tucson, Arizona. December 1983.
12. Griswold, Ralph E., Robert K. McConeghy, and William H. Mitchell. *A Tour Through the C Implementation of Icon; Version 5.9.* Technical Report TR 84-11, Department of Computer Science, The University of Arizona, Tucson, Arizona. August 1984.
13. Korb, John Timothy. *The Design and Implementation of a Goal-Directed Programming Language.* Ph.D. Dissertation, Technical Report TR 79-11, Department of Computer Science, The University of Arizona, Tucson, Arizona, June 1979.
14. Hanson, David R., and Walter J. Hansen. *Icon Implementation Notes.* Technical Report TR 79-12a, Department of Computer Science, The University of Arizona, Tucson, Arizona, February 1980.

Appendix A — The Parse Tree

The parse tree is a collection of nodes, described below, rooted at a **proc** node. Nodes have a common format: the first field contains the node type, the second and third fields contain a line and column number relating the node to the source program, and the next zero to four fields contain node-dependent information. The line and column numbers are usually those of the first token or the primary token of the construct; for example, in **binop** nodes, they are the location of the operator; in **if** nodes, they are the location of the if token.

The following list of node types gives a brief description of the node and a list of the node-dependent fields and their uses. The fields are named *val* if they contain an integer value, *str* if they contain a pointer to a string, or *tree* if they contain a pointer to another node (a leaf or subtree). A digit between 0 and 3 is appended indicating its position in the node.

Seven of the nodes — **cset**, **id**, **int**, **op**, **real**, **res**, and **str** — are leaf nodes. These nodes, allocated and returned by the lexical analyzer, represent source program tokens. The remaining nodes contain one or more pointers to other nodes, either leaves or subtrees.

activat A transmission expression (*e1* @ *e2*).
tree0 The operator (an **op** node).
tree1 *e1*.
tree2 *e2*.

alt An alternation expression (*e1* | *e2*).
tree0 *e1*.
tree1 *e2*.

augop An augmented assignment expression (*e1* ○:= *e2*).
tree0 The operator.
tree1 *e1*.
tree2 *e1*.

bar A repeated alternation expression (|*e*).
tree0 *e*.

binop A binary operation (*e1* ○ *e2*).
tree0 The operator.
tree1 *e1*.
tree2 *e2*.

break A break expression (break [*e*]).
tree0 *e*.

case A case expression (case *e* of { ... }).
tree0 *e*.
tree1 The list of case clauses. If there is only one case clause, this field points to the **ccls** node; if there are more, it points to a **clist** node.

ccls A case clause (*e1* : *e2*).
tree0 *e1*. For a default clause, this field points to a **res** node that contains the reserved word default.
tree1 *e2*.

clist A list of case clauses. The list is represented as a binary tree, with left branches pointing to case clauses and right branches pointing to a list of the remaining case clauses. The right branch of the last **clist** node points directly to a **ccls** node.
tree0 A case clause (pointer to a **ccls** node).
tree1 Pointer to another **clist** node, or to the last **ccls** node in the list.

conj A conjunction expression (*e1* & *e2*).
tree0 *e1*.
tree1 *e2*.

create A create expression (create *e*).
tree0 *e*.

cset A leaf node representing a cset literal.
str0 The string equivalent of the literal.
val1 The length of the string.

elist An expression list, as in a list construction or the argument list in a procedure call. An expression list, like a list of case clauses, is represented as a binary tree.
tree0 An expression.
tree1 Pointer to another **elist** node, or to the last expression in the list.

empty This node is used as a placeholder for missing expressions in control structures and expression lists.

field A field reference to a record (*e* . *ident*).
tree0 *e*.
tree1 Pointer to an **id** node, containing the field name *ident*.

id A leaf node representing an identifier.
str0 The name of the identifier.

if An if expression (if *e1* then *e2* [else *e3*]).
tree0 *e1*.
tree1 *e2*.
tree2 *e3*.

int A leaf node representing an integer literal.
str0 The string representation of the literal.

invok A procedure call or mutual evaluation expression (*e* (*args*)).
tree0 *e*.
tree1 The argument list *args*. If there is one argument, this field points to the expression; if there are more, it points to an *elist* node.

key A keyword reference (&*ident*).
val0 The index of the keyword *ident*, defined in the file tran/keyword.h.

limit A limitation expression (*e1* \ *e2*).
tree0 *e1*.
tree1 *e2*.

list A list ([*e1*, *e2*, ...]).
tree0 The list of elements. If there is one element, this field points to the expression; if there are more, it points to an **elist** node.

loop A loop expression (*loop e1* [do *e2*]).
tree0 The style of loop. This field points to a **res** node, which identifies the reserved word that introduced the loop.
tree1 *e1*.
tree2 *e2*.

next A next expression.

not A not expression (not *e*).
tree0 *e*.

op A leaf node representing an operator.
val0 The token type of the operator.

proc A procedure. This node is always at the root of the parse tree.

tree0 The procedure name. This field points to an **id** node containing the name.

tree1 The initial clause.

tree2 The procedure body. If there is one expression in the procedure body, this field points to it; if there are more, it points to an **elist** node.

tree3 A node containing the end token. This field is used to supply a line number for the implicit return at the end of a procedure.

real A leaf node representing a real number literal.

str0 The string representation of the literal.

res A leaf node representing a reserved word.

val0 The token type of the reserved word.

ret A return or fail expression.

tree0 The type of return. This field points to a **res** node, which contains the reserved word return or fail.

tree1 The expression following return, or a pointer to an **empty** node.

scan A scanning expression (*e1* ? *e2*).

tree0 The operator.

tree1 *e1*.

tree2 *e2*.

sect A section expression (*e1* [*e2* : *e3*]).

tree0 *e1*.

tree1 *e2*.

tree2 *e3*.

slist A list of expressions separated by semicolons, as in a procedure body (a statement list). This list, like expression lists and case lists, is represented as a binary tree.

tree0 An expression in the list.

tree1 A pointer to another **slist** node, or to the last expression in the list.

str A leaf node representing a string literal.

str0 The string value of the literal.

val1 The length of the string, necessary because the string may contain the ASCII *null* character, which would otherwise terminate the string.

susp A suspend expression (suspend [*e*]).

tree0 *e*.

toby A to-by expression (*e1* to *e2* by *e3*).

tree0 *e1*.

tree1 *e2*.

tree2 *e3*.

to A to expression (*e1* to *e2*).

tree0 *e1*.

tree1 *e2*.

unop A unary operation (○ *e*).

tree0 The operator.

tree1 *e*.

Appendix B — Icon Grammar

The following grammar describes the Icon language. Reserved words and operators are shown in a sans-serif type face; nonterminals are in italics. The nonterminals *ident*, *literal*, *strliteral*, and *empty* are left undefined in the syntax.

```
   program  →  decls

     decls  →  empty
            →  decls decl

      decl  →  record
            →  proc
            →  global
            →  link

      link  →  link lnklist

   lnklist  →  lnkfile
            →  lnklist , lnkfile

   lnkfile  →  ident
            →  strliteral

    global  →  global idlist

    record  →  record ident ( arglist )

      proc  →  prochead ; locals initial procbody end

  prochead  →  procedure ident ( arglist )

   arglist  →  empty
            →  idlist

    idlist  →  ident
            →  idlist , ident

    locals  →  empty
            →  locals retention idlist ;

 retention  →  local
            →  static
            →  dynamic

   initial  →  empty
            →  initial expr ;

  procbody  →  empty
            →  nexpr ; procbody

     nexpr  →  empty
            →  expr

      expr  →  expr1a
            →  expr & expr1a

    expr1a  →  expr1
            →  expr1a ? expr1
```

```
expr1  →  expr2
       →  expr2 op1 expr1
       →  expr2 op1a expr1
       →  expr2 ?:= expr1
       →  expr2 &:= expr1
       →  expr2 @:= expr1

  op1  →  := | :=: | <- | <->

 op1a  →  +:= | -:= | *:= | /:= | %:= | ^:= | ++:= | --:= | **:= | ||:= | |||:=
       →  <:= | <=:= | =:= | >=:= | >:= | ~=:=
       →  <<:= | <<=:= | ==:= | >>=:= | >>:= | ~==:=
       →  ===:= | ~===:=

expr2  →  expr3
       →  expr2 to expr3
       →  expr2 to expr3 by expr3

expr3  →  expr4
       →  expr4 | expr3

expr4  →  expr5
       →  expr4 op4 expr5

  op4  →  < | <= | = | >= | > | ~=
       →  << | <<= | == | >>= | >> | ~==
       →  === | ~===

expr5  →  expr6
       →  expr5 op5 expr6

  op5  →  || | |||

expr6  →  expr7
       →  expr6 op6 expr7

  op6  →  + | - | ++ | --

expr7  →  expr8
       →  expr7 op7 expr8

  op7  →  * | / | % | **

expr8  →  expr9
       →  expr9 ^ expr8

expr9  →  expr10
       →  expr9 \ expr10
       →  expr9  @ expr10

expr10 →  expr11
       →  not expr10
       →  @ expr10
       →  | expr10
       →  op10 expr10

 op10  →  . | ! | + | - | ~ | = | * | / | \ | ? | ^
```

```
  exprl1  →  ident
          →  literal
          →  & ident
          →  exprl1 . ident
          →  exprl1 [ nexpr ]
          →  exprl1 ( exprlist )
          →  exprl1 { exprlist }
          →  [ exprlist ]
          →  ( exprlist )
          →  { compound }
          →  while
          →  until
          →  every
          →  repeat
          →  next
          →  break nexpr
          →  create expr
          →  if
          →  case
          →  return
          →  section

   while  →  while expr
          →  while expr do expr

   until  →  until expr
          →  until expr do expr

   every  →  every expr
          →  every expr do expr

  repeat  →  repeat expr

      if  →  if expr then expr
          →  if expr then expr else expr

    case  →  case expr of { caselist }

caselist  →  cclause
          →  caselist ; cclause

 cclause  →  default : expr
          →  expr : expr

  return  →  fail
          →  return nexpr
          →  suspend nexpr

 section  →  exprl1 [ expr sectop expr ]

  sectop  →  : | +: | -:

exprlist  →  nexpr
          →  exprlist , nexpr

compound  →  nexpr
          →  nexpr ; compound
```

Appendix C — Ucode

The intermediate ucode generated by the Icon translator resembles a stack-oriented assembly language. A ucode program is a sequence of labels and instructions. A label marks a location in the program to which other instructions may transfer control. Labels are of the form "**lab L***n*", where *n* is a decimal number. A ucode instruction either describes an imperative operation or communicates information to the Icon linker. Instructions consist of an opcode followed by zero or more arguments. Arguments can be decimal or octal integers, names, or label references.

The intermediate language operates exclusively on the stack. There are several kinds of objects that can appear on the stack: descriptors, which represent Icon values and variables; procedure frame markers, which mark the beginning of a new procedure frame; expression frame markers, which delimit expression instances; and generator frame markers, which mark inactive generators. For more details about the stack, refer to Section 3.2.

The opcodes and their arguments are described in three groups below. The global symbol table file has a similar format. The opcodes used there are described in the fourth group.

Imperative Instructions

The instructions below, together with the operators described in the next section, represent run-time actions for which code is executed.

bscan

Save the values of &subject and &pos on the stack and establish values for them. This operation is reversible.

ccase

Duplicate the value on the stack just below the current expression frame. Used in case expressions.

chfail *lab*

Change the failure label for the current expression frame to *lab*. Used for repeated alternation.

coact

Switch co-expression evaluation. Create a procedure frame on the current co-expression stack. Transfer the result from old stack to new stack, dereferencing if necessary. Set the activator field in new stack block to point to old co-expression stack block. Return from procedure frame on new co-expression stack.

cofail

Fail from current co-expression to activating co-expression. Create a procedure frame on current co-expression stack. Fail from procedure frame on activator's co-expression stack.

coret

Increment the result count field in the current co-expression stack block. Switch evaluation to activating co-expression. Create a procedure frame on current co-expression stack. Transfer the result from old stack to activator's stack, dereferencing it if the result is on the old stack. Return from the procedure frame on new co-expression stack.

create

Create a co-expression. Allocate co-expression stack and heap blocks. Copy the arguments and locals variables from the current procedure frame into the heap block. Create a procedure frame in the new co-expression stack using the arguments and other locals from current procedure frame. Create a procedure frame for dummy call to coact on the new co-expression stack. Push a descriptor representing the new co-expression onto current co-expression stack.

cset *n*

Push a descriptor representing the cset literal at constant table location *n* onto the stack.

dup

Push a descriptor representing the null value onto the stack, and then duplicate the value that was the previous top of the stack. Used in most augmented assignments.

efail

Signal failure in the current expression. If there are any inactive generators, reactivate the most recent one. If there are none, branch to the failure label for the current expression frame. If the failure label is null, exit the current expression frame, and signal failure in the enclosing one.

eret

Return a value from an expression. Save the value on top of the stack, exit the current expression frame, and push the value onto the stack as part of the enclosing expression frame.

escan

Restore &subject and &pos from the stack. This operation is reversible.

esusp

Suspend a value from an expression. The value on the top of the stack is saved, and a generator frame hiding the current expression frame is created. The surrounding expression frame is duplicated, and the value is pushed onto the stack as part of that expression frame. When reactivated, **esusp** in turn reactivates any inactive generators in the suspended expression.

field *name*

Access the field *name* of the record on the top of the stack.

file *name*

Set the file name to *name* for use in error messages and tracing. Used at the beginning of each procedure.

goto *lab*

Transfer control to the instruction following label *lab*.

init *lab*

If the initialization statement for the current procedure has already been executed once, go to *lab*.

int *n*

Push a descriptor representing the integer literal at constant table location *n* onto the stack.

invoke *n*

Invoke a procedure or create a record. The number of arguments or fields on the stack is given by *n*. The procedure (which may be a record constructor) is on the stack, just beyond the arguments. After invocation, the arguments are popped from the stack, and the returned value is pushed (see **pret**).

keywd *n*

Push a descriptor representing a value or trapped variable representing keyword *n* onto the stack. (See tran/keyword.h for keyword numbers.)

limit

Check the value on the top of the stack for a legal limitation value. If the value is zero, failure is signaled in the current expression (see **efail**).

line *n*

Set the line number to *n* for use in error messages and tracing.

llist *n*

Create a list of *n* values. The values are popped from the stack and the created list is pushed back onto the stack.

lsusp

Decrement the limitation counter for the current expression frame. If the counter becomes zero, then return a value from the current expression frame (see **eret**); otherwise, suspend a value from the current expression frame (see **esusp**).

mark *lab*

Save the current expression and generator frame pointers on the stack, then create a new expression frame, with failure label *lab*. Control is transferred to *lab* if failure occurs in the expression frame and there are no suspended generators to reactivate (see **efail**). The failure label L0 indicates that control is to be transferred to the failure label in the enclosing expression.

pfail

Return from the current procedure, and signal failure (see **efail**).

pnull

Push a descriptor representing the null value onto the stack.

pop

Pop the top element off of the stack.

pret

Return from the current procedure with the result that is on top of the stack.

psusp

Suspend from the current procedure with the result that is on top of the stack.

push1

Push a descriptor representing the integer 1 onto the stack.

pushn1

Push a descriptor representing the integer -1 onto the stack. This is used as default in mutual goal-directed evaluation.

real *n*

Push a descriptor representing the real literal at constant table location *n* onto the stack.

refresh

Allocate space for a new co-expression stack. Create a procedure frame in new co-expression stack using arguments and other locals from entry block for co-expression operand. Create a procedure frame for dummy call to coact on new co-expression stack. Push a descriptor representing the new co-expression onto current co-expression stack.

sdup

Duplicate the descriptor on the top of the stack. Used in assignment augmented with string scanning.

str *n*

Push a descriptor representing the string literal at constant table location *n* onto the stack.

unmark *n*

Exit from *n* expression frames. No value is pushed onto the stack in their place.

var *n*

Push the descriptor for the variable at location *n* in the local symbol table onto the stack.

Operators

The instructions below perform the functions corresponding to the indicated Icon operator. The operands are evaluated and pushed onto the stack from left to right, so that the topmost element of the stack is the rightmost operand. The operands are popped before the result of the operation is pushed onto the stack. All operations dereference their operands as necessary, but only after all operands have been evaluated and pushed onto the stack. All operations attempt to convert their operands to an appropriate type. If this implicit conversion fails, an error is issued. Relational tests fail if the specified condition is not met; the result of a successful comparison is the value of the right-hand operand. Arithmetic operations cause an error to be issued if the result overflows or underflows. If an operation cannot be performed for some other reason, it fails.

<table>
<tr><td><b>asgn</b></td><td>x := y</td><td><b>null</b></td><td>/x</td></tr>
<tr><td><b>bang</b></td><td>!x</td><td><b>number</b></td><td>+x</td></tr>
<tr><td><b>cat</b></td><td>x || y</td><td><b>numeq</b></td><td>x = y</td></tr>
<tr><td><b>compl</b></td><td>~x</td><td><b>numge</b></td><td>x >= y</td></tr>
<tr><td><b>diff</b></td><td>x -- y</td><td><b>numgt</b></td><td>x > y</td></tr>
<tr><td><b>div</b></td><td>x / y</td><td><b>numle</b></td><td>x <= y</td></tr>
<tr><td><b>eqv</b></td><td>x === y</td><td><b>numlt</b></td><td>x < y</td></tr>
<tr><td><b>inter</b></td><td>x ** y</td><td><b>numne</b></td><td>x ~= y</td></tr>
<tr><td><b>lconcat</b></td><td>x ||| y</td><td><b>plus</b></td><td>x + y</td></tr>
<tr><td><b>lexeq</b></td><td>x == y</td><td><b>power</b></td><td>x ^ y</td></tr>
<tr><td><b>lexge</b></td><td>x >>= y</td><td><b>random</b></td><td>?x</td></tr>
<tr><td><b>lexgt</b></td><td>x >> y</td><td><b>rasgn</b></td><td>x <- y</td></tr>
<tr><td><b>lexle</b></td><td>x <<= y</td><td><b>rswap</b></td><td>x <-> y</td></tr>
<tr><td><b>lexlt</b></td><td>x << y</td><td><b>sect</b></td><td>x[y:z]</td></tr>
<tr><td><b>lexne</b></td><td>x ~== y</td><td><b>size</b></td><td>*x</td></tr>
<tr><td><b>minus</b></td><td>x - y</td><td><b>subsc</b></td><td>x[y]</td></tr>
<tr><td><b>mod</b></td><td>x % y</td><td><b>swap</b></td><td>x :=: y</td></tr>
<tr><td><b>mult</b></td><td>x * y</td><td><b>tabmat</b></td><td>=x</td></tr>
<tr><td><b>neg</b></td><td>-x</td><td><b>toby</b></td><td>x to y by z</td></tr>
<tr><td><b>neqv</b></td><td>x ~=== y</td><td><b>unions</b></td><td>x ++ y</td></tr>
<tr><td><b>nonnull</b></td><td>\x</td><td><b>value</b></td><td>.x</td></tr>
</table>

Non-Imperative Instructions

The following instructions generate no executable code. Instead, they communicate various information to the linker each procedure and its symbol table. An Icon procedure is translated into a sequence of ucode instructions beginning with a **proc** instruction, followed by a sequence of **local** instructions, a sequence of **con** instructions, a **declend** instruction, then the imperative instructions describing the procedure body. An **end** instruction terminates the procedure.

proc *name*

Begin a new procedure with the indicated name. The local and constant tables are initialized. The procedure block is not generated at this time, since the local identifiers have not yet been declared.

local *n, flags, name*

Enter *name* into the current procedure's local symbol table at location *n*. The symbol's *flags* indicate information about the symbol, its scope, and its retention. All identifiers referred to in a procedure appear in the local symbol table. If an identifier is undeclared, its scope is determined by consulting the global symbol table and a list of functions.

con *n, flags, value*

Enter *value* into the current procedure's constant table at location *n* in the table. The type of the constant (integer, real, or string) is indicated by *flags*. For integer and real literals, *value* is an 11-digit octal number; for string and cset literals, it is a comma-separated list of 3-digit octal numbers, each representing one byte in the string.

declend

Signal the end of the procedure prologue. The procedure block is generated at this point.

end

Signal the end of a procedure.

Global Symbol Table Instructions

A global symbol table file is output during each translation. Record declarations appear first in the file; they are output as they are encountered in the Icon source program. The first instruction following the record declarations is **impl**, which may be followed by a **trace** instruction, then by the global declarations. The global declarations are output at the end of translation.

record *name,n*

Declare a record with the indicated name and *n* fields. One line for each field follows this line, each containing the field number and name.

impl *scope*

Declare the implicit scope as indicated. *Scope* can be either **local** or **error**. If the implicit scope is **error**, undeclared identifiers are flagged as warnings during linking; otherwise, they are made local variables. The implicit scope is **error** if the **-u** option was given on the translator command line, otherwise it is **local**.

trace

Enable run-time tracing. This instruction is present if the **-t** option was given on the translator command line, and causes the keyword **&trace** to be initialized to -1.

global *n*

Begin the global symbol table. There are *n* global declarations following, one per line. Each global declaration contains a sequence number, the flags, the identifier name, and the number of formal parameters (for procedures) or fields (for records).

link *name*

Search each directory named in the *IPATH* environment variable for a file named *name.ul*. If the file is located, it is added to the list of files to link.

Appendix D — Data Representations

Descriptor Formats

The figures below depict each of the six descriptor types mentioned in Section 3.1. Each descriptor is two words long. The first word, or d-word, contains flags in the most significant bits. The least significant bits contain a type code, except in the case of qualifiers and variable descriptors.

Qualifier

<table>
<tr><td>0</td><td><i>length</i></td></tr>
<tr><td colspan="2"><i>address of string</i></td></tr>
</table>

Null

<table>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td></td><td><i>type</i> =0</td></tr>
<tr><td colspan="6"></td></tr>
</table>

Integer

<table>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td></td><td><i>type</i> = 1</td></tr>
<tr><td colspan="6"><i>integer</i></td></tr>
</table>

Pointer

<table>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td></td><td><i>type</i></td></tr>
<tr><td colspan="6"><i>address of data block</i></td></tr>
</table>

Variable

<table>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td></td><td><i>offset</i></td></tr>
<tr><td colspan="6"><i>address of descriptor</i></td></tr>
</table>

Trapped Variable

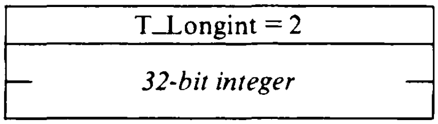
<table>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td></td><td><i>type</i></td></tr>
<tr><td colspan="6"><i>address of data block</i></td></tr>
</table>

Note: The offset in a variable descriptor is the number of *words* from the top of the block in which the descriptor that is pointed to occurs.

Data Block Formats

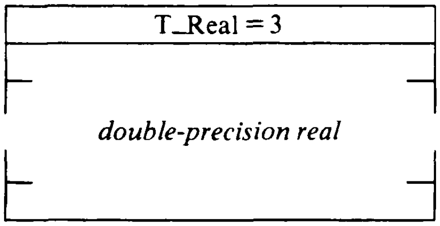
The data blocks used by the Icon system are pictured below. The first word of a block, known as its title, contains a data type code, shown as both a mnemonic and an integer, which is the same as the type code in the d-word of the descriptor that points to it.

Long Integer Block

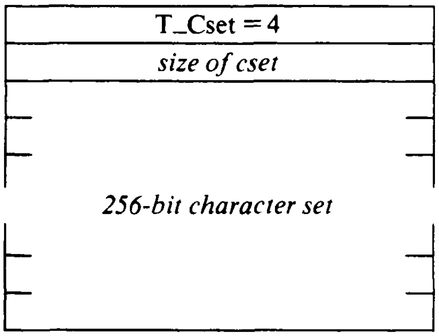


Note: Long integers are used only on computers for which sizeof(int) != sizeof(long).

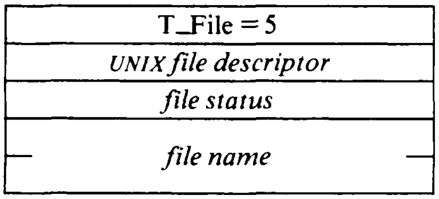
Real Block



Cset Block



File Block



Procedure Block

T_Proc = 6
size of block
entry point address
number of arguments
number of dynamic locals
number of static locals
index of first static local
procedure name
name of first identifier
⋮
name of last identifier

Notes: Identifiers include arguments as well as local identifiers. Similar blocks are used for built-in functions; in this case the word for the number of dynamic locals contains -1. For functions, there are no argument names. Functions like write, which have an arbitrary number of argument, are indicated by the value -1 in place of the number of arguments. Record constructors are distinguished from other functions by the value -2 in place of the number of dynamic locals. Each record declaration is distinguished by a unique record identification number, which appears in place of the number of static locals.

List Block

T_List = 7
size of list
first list element block
last list element block

List Element Block

T_Lelem = 11
size of block
number of slots in this block
index of first slot used
number of slots used
previous list element block
next list element block
first slot
⋮
last slot

Table Block

T_Table = 8
size of table
default value
first slot
⋮
last slot

Table Element Block

T_Telem = 10
hash number
next table element block
entry value
assigned value

Table Element Trapped Variable Block

T_Tvtbl = 14
hash number
table block
entry value

Note: The last descriptor in a table element trapped variable block is not used until the element is inserted in the table, at which time the table element trapped variable is converted into a table element block.

Set Block

T_Set = 20
size of set
first slot
. . .
last slot

Set Element Block

T_Selem = 21
hash number
next set element block
value

Record Block

T_Record = 9
size of block
record constructor
first field of record
. . .
last field of record

Keyword Trapped Variable Block

T_Tvkywd = 13
assignment function
value of keyword
name of keyword

Substring Trapped Variable Block

T_Tvsubs = 12
length of substring
relative position of substring
variable containing substring

Co-Expression Stack Block

· · · *stack* · · ·
T_Estack = 18
most recent activator
stack base
stack pointer
address pointer
Icon/C boundary
number of results produced
refresh block

Note: On computers with downward growing stacks, the stack precedes the title of the block, as indicated.

Refresh Block

T_Refresh = 19
size of block
entry point address
number of arguments
number of locals
procedure block
value of first identifier
⋮
value of last identifier

Note: Identifiers include arguments as well as local identifiers.