A Tour Through the C Implementation of Icon*

Cary A. Coutant

Stephen B. Wampler

TR 80-9

ABSTRACT

This paper documents the C implementation of Version 3 of the Icon programming language. The three major parts of the implementation — the translator, the linker, and the runtime system — are described. An additional section discusses how the implementation may be modified for new language features.

June 1980

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS79-03890.

A Tour Through the C Implementation of Icon

Introduction

This paper describes an implementation of Version 3 of the Icon programming language [1] for PDP-11 computers running under UNIX.* This implementation is intended to be portable to other systems with C compilers, but this is not a primary goal. The major objectives are an efficient implementation and ease of modification. The implementation does, however, share much of its design with the portable Ratfor implementation [2].

The implementation consists of three parts: the translator, the linker, and the runtime system. The translator converts an Icon source program into an intermediate code, called *ucode*. The linker† combines separately translated ucode files and produces PDP-11 assembly language, which is then assembled and loaded with the runtime system to form an executable object program.

This paper is intended to be used in conjunction with the source listings of the Icon system, although a general overview of the system can be obtained from this document alone.

1. Translator

The Icon translator is written entirely in C [3]. The translator builds a parse tree for each Icon procedure, then traverses the tree to generate code. The translator consists of eleven files of source code and nine header files. Three of the eleven source files 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 [4].

The translator produces two output files, both of which are processed by the linker: a file containing the entire global symbol table, and a file of intermediate code.

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 Lexical Analyzer

The lexical analyzer reads the Icon source program, breaking 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.

Four source files comprise the lexical analyzer: *lex.c*, *char.c*, *optab.c*, and *toktab.c*. The last 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 moves 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 indicates one of the five "primitive" types identifier, integer literal, real literal, string 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 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.

---

*UNIX is a trademark of Bell Laboratories.

†The term *linker* is actually a misnomer, so named for historical reasons. More appropriately, it is a second pass in the translation process.

The lexical analyzer finds the next token by skipping all white space and comments. The first character of the new token then indicates that it belongs to one of four classes. A letter or underscore begins an identifier or reserved word; a digit begins an integer or real literal; a single or double quote begins a string literal; any other character is assumed to begin an operator. An identifier or reserved word is completed by gathering all subsequent letters, digits, and underscores; all upper case letters are folded into lower case. The reserved word table is consulted to determine if the token is an identifier or a reserved word. A numeric literal is recognized by a finite-state automaton, which distinguishes real from integer literals by the presence of a decimal point or the letter "e". A string 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, and 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 token table is initialized in the file *toktab.c*. The table is divided into three sections: primitive types, reserved words, and operators. The primitives types are fixed in the first five 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 effectively hashes the search using the first letter as the hash value. The search needs only to examine all reserved words that begin with a single letter.

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 one 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 (the input symbol field is 0) is used. The recognizer then performs one of the following actions, depending on the value of the action field: goto the new state indicated by the value field and get the next input character; issue an error; return the value field as a pointer to the token table entry for the operator or return the value field, but push 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, reserved word index, and operator table are automatically constructed by the SNOBOL4 program *mktoktab.4*. This program reads the 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 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 Parser

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

The file *ucon.g* contains, in addition to the grammar, 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 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 intermediate code 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.

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 several 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**, and **every-do** 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 [5]. 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 *err.h* that is associated with the current parser state. This error table must currently be constructed by hand from the verbose listing of parser states obtained by running Yacc with the **-v** option.

1.3 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 code in the intermediate language, *ucode*. Appendix C contains a description of the intermediate language.

The file *code.c* contains both the tree node allocation and the code generation routines. There are two include files: *code.h* contains macros and definitions needed by the code generator, while *tree.h* defines the tree nodes and the macros that allocate them. 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 two arguments. The first argument is a pointer to the root of a tree or subtree for which code is to be generated. The second argument is the failure label, discussed below. 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

runtime library routine, and finally tests for failure.

Since the intermediate language and its underlying implementation are stack-oriented, all language operations produce one value (which may be the null value). The code generated for each operation has a net effect of pushing a value onto the stack. For example, the code for the binary operators pushes the two operands onto the stack, then calls the library routine for the operator. The library routine uses the two operands, pops them off the stack, and pushes the result onto the stack. Thus, the net effect of the binary operation is to push one value onto the stack.

The second argument to *traverse*, the current failure label, is always the label of the **drive** instruction at the end of the current "driven expression". A driven expression is one within which backtracking is contained. As defined by the language, all control structures (except **every**) contain their constituent expressions, as do the semicolons in compound expressions. Driven expressions appear in the grammar as the non-terminals *dexpr* or *ndexpr*. A **drive** node represents a driven expression, and causes a **mark/drive** pair of instructions to surround the code generated by its subtree. Any failure within a driven expression causes a branch to the **drive** instruction, which drives the expression for alternatives. Wherever code that might cause failure is generated, a test for failure is generated that branches to the failure label. For most node types, the failure label for its subtrees is the same as that for itself. For any node type that generates a **mark/drive** pair of instructions, a new failure label is used for the surrounded code, and this label is emitted just before the **drive**.

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** node (expression list), *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 literal lists, which need to know the number of arguments that will be pushed onto the stack.

Two stacks are used by *traverse* — a loop stack and a case stack. The loop stack contains the break and next labels for loops. For each loop expression, the code generator allocates a break and a next label, and pushes these labels onto the loop stack. The code for **break** and **next** nodes branches to the appropriate label from the top of this 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 statement.

1.4 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 table and is automatically constructed from a keyword list 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 may 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 undeclared identifiers. The literal table contains entries for literal strings, integers, and floating-point constants.

Both the local and literal tables are associated with the current procedure being parsed, and are written to the ucode 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 global declarations 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 name 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. An unfortunate 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 should not produce an unreasonable number of duplicate constants.

A final task of the symbol table manager is the identification of keywords. The symbol table manager maintains a list of the legal keywords and, upon request, returns a numeric identification for a keyword identifier to the parser. An automatic procedure exists for creating the keyword table: the SNOBOL4 program *mkkeytab.4* reads a list of keywords from the file *keywords* and produces the keyword table in *keywords.c*. The file *keywords* is simply a list of the keywords and a numeric identification for each. Since the number of keywords is small, and only a few references to keywords are typical in an Icon program, lookup in the keyword 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. Some thought has been given to allowing automatic expansion of the symbol table on overflow, but this enhancement has been omitted from the current version.

2. Linker

The linker performs three tasks: combining the global symbol tables from one or more runs of the translator, resolving undeclared identifiers, and translating the ucode to assembly code. The first task is done first; 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 intermediate code file. A single file of assembly code is produced.

The linker consists of eight files of C source code and four header files. The symbol table module, in the file *sym.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 *lex.c*, recognizes the instructions in both the global symbol table files and the intermediate code files. The global symbol tables are merged by the routine in *glob.c*, and the intermediate code files are produced by the routines in *code.c*. Of the remaining source files, *ulink.c* and *mem.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 procedures (or *functions*) and the ucode operations, respectively.

The first phase of the linker consists of reading the global symbol table file from each translator run, and entering 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 runtime when referencing a field.

The second phase reads each intermediate code file in sequence, emitting assembler code as each procedure is encountered. Appendix C describes the intermediate code. The intermediate code contains a prologue for each procedure, beginning with a **proc** opcode, followed by a series of **loc** opcodes describing the local symbol table, a **locend** opcode terminating the local symbol table, and a series of **con** opcodes describing the constant table. 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: (1) if the identifier has been entered in the global symbol table, it is entered into the local symbol table as a global variable; else (2) if the identifier matches the name of a function, it is entered into the local symbol table as a function; else (3) it is entered as a local variable and a warning is issued if the linker was run with the **-u** option. The constant table contains an entry for each literal used in the procedure.

Once the prologue has been processed, a procedure data block (see Section 3.1) is emitted into the assembler code. The initial value of the procedure variable has type *procedure* and will point to this block.

Opcodes following the prologue represent execution-time operations, and cause code to be emitted. Most assembler code is emitted through the routine *emit*, which outputs code according to several *templates*. This routine is called with an arbitrary number of arguments, a list of *template calls*. Each template call is a template name, defined in the file *code.h*, followed by parameters to that template. The last template call must be followed by a 0 to indicate the end of the argument list to *emit*. For example, the following code, taken from the processing for the **mark** opcode, causes the assembler output shown to be emitted.

```
emit(C_MOVI, -1, "_usignal",
     C_PUSH, "r4",
     C_MOV, "sp", "r4",
     C_PUSH, "r3",
     C_CLR, "r3",
     0);
```

```
mov     $-1,_usignal
mov     r4,-(sp)
mov     sp,r4
mov     r3,-(sp)
clr     r3
```

The **end** opcode signals the end of a procedure, and causes the linker to emit data blocks for real numbers and long (32-bit) integers in the procedure's constant table. Literal references to these data types generate code that builds a descriptor (see Section 3.1) that points to these blocks. References to short (16-bit) integer literals generate code that builds a descriptor containing the value. References to string literals generate code that builds a descriptor pointing into the identifier table (see below).

When all the intermediate code files have been processed, the linker emits procedure data blocks for all record constructors and functions, followed by the record/field table, initial values and names for all global and static variables, then the identifier table.

The record/field table is a doubly-indexed table, 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 variables is the null value unless the global variable is a procedure, function, or record constructor, in which case the initial value is a descriptor of type *procedure* pointing to the appropriate procedure data block. The values output use the data representations described in Section 3.1.

The names of global and static variables are output as *string qualifier* descriptors (see Section 3.1), and are used by the function *display*. All string qualifiers contained in the generated procedure data blocks and global and static names point into the identifier table, which is just a static string space for that purpose.

3. Runtime System

The runtime library is a collection of routines that collectively provide support for the execution of an Icon program. This library is searched by the loader for those routines necessary for a particular Icon program. The assembly code generated by the linker contains subroutine calls to library routines to perform most high-level operations where in-line code would be inappropriate. An executable program is created by assembling the linker output, then loading a startup routine and the assembler output with the runtime library and a tailored version of the C library. The startup routine, runtime library, and C library together form the runtime system.

The runtime library has a two-level structure. The top level, *lib*, contains routines which relate directly to source language operations. For example, *plus* performs addition and *invoke* performs procedure invocation. Underneath *lib* is *rt*, which contains routines for performing common operations needed by many routines in *lib*. In particular, *rt* contains routines that handle storage allocation and reclamation, type conversion, data comparison, integer arithmetic with overflow checking, program initialization, generator suspension, and tracing.

Well over 90 percent of the runtime system is coded in C; the remainder is coded in assembly language. Of the routines coded in assembly language, one is the startup routine, one does integer arithmetic with overflow checking (C does not provide this), and the rest modify the stack in ways that C does not allow.

3.1 Data Representations

Icon has two elementary forms of data objects — values and variables. Values can often be converted from one data type to another; when done automatically, this is called *coercion*. There are three kinds of variables, each discussed below: *natural variables*, *created variables*, and *trapped variables*. The process of obtaining the value referred to by a variable is called *dereferencing*.

In this implementation of Icon, all data objects are represented by a two-word *descriptor*, which may, depending on the type of the object, refer to some other area of memory for the actual value. The first word of the descriptor always indicates the data type, and the second word either contains the value or a pointer to it. There are six descriptor formats, pictured in Appendix D: *null*, *string qualifier*, *short integer*, *value*, *variable*, and *trapped variable*. These formats are distinguished from one another by the first few bits of the first word (except that a *null* descriptor is distinguished from a *string qualifier* only by the contents of the second word). Among *short integer*, *value*, and *trapped variable* descriptors, the low-order six bits of the first word identify the type of object represented; the remaining bits in the first word contain flags classify the object as numeric, integer, aggregate (e.g., list, table, stack), and whether or not the second word is a pointer (historically, a "floating address"[6]).

The *null* descriptor represents the null value. A *string qualifier* represents a string, and contains the length of the string and a pointer to the first character of the string. A *short integer* descriptor represents an integer small enough to fit in the second word of the descriptor; all larger integers are represented by a *value* descriptor, which represents values of all data types other than string and null. The *value* descriptor contains a pointer to a *data block* of appropriate format for a value of the given type. The data block formats for each data type are shown in Appendix D.

A *variable* descriptor represents either a natural variable or a created variable. A natural variable contains a pointer to a descriptor at a fixed location (for a global variable) or a location on the stack (for a local variable) where the value of the variable is stored. A created variable, formed by a table or list reference, contains a pointer to a descriptor in a table or list block, where the referenced element is located. Since table and list elements are often in the heap, created variables also contain an offset which indicates the distance (in words) from the beginning of the data block to the referenced descriptor. This offset is used during the marking phase of garbage collection, discussed in Section 3.3.

A *trapped variable* [7] descriptor represents a variable for which special action is necessary upon dereferencing or assignment. Such variables include substrings, non-existent elements of open lists and tables, and certain keyword variables. Each type of trapped variable is distinguished by the first word of the descriptor.

Substring trapped variables, created by a section or subscripting operation, contain a pointer to a data block which 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, taking the substring of a non-variable produces a *string qualifier*.

Table and list element trapped variables, formed by referencing a non-existent element of an open table or an element one position beyond the end of an open list, similarly contain a pointer to a data block that contains enough information for assignment to add the element to the referenced list or table.

Trapped variables for the keywords *&pos*, *&trace*, and *&random* need no additional information. It is sufficient to know the type of trapped variable on dereferencing — the value of the keyword can be accessed and returned. On assignment, the new value is coerced to integer type, checked for validity, and assigned to the keyword. The trapped variable for the keyword *&subject* is similar to a substring trapped variable, except that the original variable is unnecessary. This trapped variable is used only when a substring of *&subject* is formed by the function *move*, *tab*, or *insert*, or by the prefix = operator. Assignment to a subject trapped variable causes coercion of the new value to string type, and an automatic assignment to *&pos*.

Strings formed during program execution are placed in the *string space*; string qualifiers for these strings point into this region. Substrings of existing strings are not allocated again; instead, a string qualifier is formed that points into the existing string. When storage is exhausted in the string space, 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 *heap*. Data blocks have a rigid format dictated by the garbage collection algorithm. The first word of the block always contains a type code which identifies the structure of the rest of the block. Blocks that contain pointers to other blocks always use *variable* descriptors for the pointers, and the descriptors always 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 heap, the garbage collector is invoked to reclaim unused space and compact the heap; if enough space cannot be reclaimed, the heap is expanded if possible.

3.2 Stack Organization

The system stack is the focus of activity during the execution of an Icon program. All operators, built-in functions, and Icon procedures expect to find their arguments at the top of the stack, and replace the arguments with the result of their computation. Local variables for Icon procedures are also kept on the stack. The arguments, local variables, 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. Appendix E summarizes the layouts of all the stack frames.

Before an Icon procedure calls another Icon procedure, the caller pushes the procedure to be called (a descriptor – procedures are data objects in Icon) onto the stack. The caller then pushes each argument (also a descriptor) onto the stack, leftmost argument first. Since the stack starts in high memory and grows downward, the arguments appear on the stack in reverse order. 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 runtime library routine *invoke* is then called, which checks that the first descriptor pushed above actually does represent a procedure or a variable whose value is a procedure. This descriptor points to a procedure data 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. *Invoke* next adjusts the number of arguments supplied to match the number expected, deleting excess arguments or supplying the null value for missing ones. 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 variable. *Invoke* then transfers control to the procedure's entry point, and execution of the Icon program resumes in the new procedure.

When a procedure is ready to return to its caller, it pushes its return value (a descriptor) on the stack. It then transfers control to the routine *uret*, 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 heap or string space 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 much 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, since the number of arguments expected by a function is not fixed. Instead, the field in the procedure data block that indicates the number of arguments expected contains -1, which identifies the procedure as a built-in function to *invoke*. It also does not push local variable initializations for functions since the C procedure allocates its own stack space. C procedures have an entry sequence that creates a new procedure frame; since the *invoke* routine has already done this, the entry point for functions is four bytes past the actual beginning of the code (the entry sequence consists of a 4-byte **jsr** instruction).

Functions are written with one argument, *nargs*, which corresponds to the word that contains the number of arguments supplied. A macro, *ARG(n)*, is available that uses the address and contents of this word to calculate the location of 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 checking that arguments were actually supplied, for supplying defaults for missing arguments, and for dereferencing arguments that are variables. Because of the calling protocol, *ARG(0)* accesses the location where the return value should be stored. Functions must place their result there, then return through normal C conventions, which transfers control to the routine *cret*. This routine merely restores the previous procedure state, and returns control to the calling Icon procedure, where the arguments are popped.*

Operators are written like functions, with two exceptions. The syntax demands a certain number of arguments, so operators always have the correct number of arguments. Also, since operators are not variables (as function and procedure names are), the name of the operator is known at translation time, and the Icon procedure calls it directly (at its normal entry point) without going through *invoke*. Thus, there is no procedure-typed descriptor on the stack referring to the operator, and the proper place to return a value is in *ARG(1)*. Although the *nargs* argument is not strictly necessary for operators, the convention was preserved for uniformity.

When an operator or function fails to produce a value, it sets the global variable *usignal* to zero. The descriptor for the return value is still on the top of the stack, but its value is meaningless. Icon procedures check *usignal* after every operator that can fail, and after every function or procedure call. (The translator does not know whether a given function or procedure can fail, so it assumes that it can.) If *usignal* is zero, then failure has occurred, and the current "driven expression" fails (unless dormant generators exist, as described below).

Driven expressions (see Section 1.3) are evaluated within an *expression frame*. When the evaluation of a driven expression is complete, whether it has produced a result or failed, the expression frame must be popped

*The Icon procedure has no way of knowing that it was calling a function. It contains an instruction, immediately after the **jsr** to *invoke*, that pops all the supplied arguments from the stack. If the procedure being called was not a function, then *uret* pops the arguments, and returns to the instruction following the stack pop. If the procedure was a function, then *cret* returns to the instruction that does the pop.

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 would normally 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 the backtracking: backtracking is restricted in the language to the current driven expression only.

When evaluation of a driven 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. When exiting the expression frame, the result of the expression, 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 the expression failed, the actions are the same, except that the descriptor for the result contains no meaningful value.

If an expression has any generators, then there is a *generator frame* within the current expression frame for each generator that is dormant (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 (went dormant). If a failure occurs in a driven expression, control transfers to code at the end of the expression, which calls the runtime library routine *drive*. The *drive* routine examines the generator frame pointer to see if there are any dormant generators within the current expression frame. If there are not, the expression frame is exited as described above; if there are, the stack is restored to the state preserved in the most recent generator frame, and the generator is reactivated.

A generator suspends itself by calling the runtime library routine *save*. This routine preserves the state of the stack by duplicating the current expression frame, bounded on one end by the current expression frame (excluding the expression marker), on the other end by the top of the stack. A generator marker is pushed onto the stack, followed by the duplicate expression frame. The *save* routine then returns to the suspending generator with the value 1. The generator then performs a normal return sequence.

When reactivated by *drive*, 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 *save* was called. *Drive* then returns to the generator with the value 0, as if the call to *save* had returned 0. Thus, the following outline is typical of operators and functions that generate a sequence of values.

```
compute first value;
while (not exhausted) {
   if (save()) {
      store return value;
      return;
      }
   compute next value;
   }
usignal = 0;
return;
```

The effect of driving an expression containing generators is that *save* returns 1 on its initial call, and the generator returns a value. If alternatives are needed, backtracking occurs, and the effect is, as far as the generator can tell, that *save* has returned 0, not 1, and the generator computes the next value, and suspends with that value. When the generator is exhausted, it merely fails without suspending, which just passes the failure back to the next most recent dormant generator, if any.

The alternation operator is handled only slightly differently. The *save* routine assumes that the operator, procedure, or function that called it is suspending. This is not the case with alternation, so the runtime library routine *csave* is called to handle alternation. This routine does nothing but call *save*, and relays the return value back to the Icon procedure. The Icon procedure then skips the alternative if the return value was 1, or evaluates the alternative if the return value was 0.

There is one other type of frame — a *loop frame*. When a loop is entered, a *loop marker* is pushed onto the stack, and the *loop frame pointer* is set to point to it. The loop marker saves the values of all three frame pointers, so they can be restored upon **break** or **next**. For **every** loops with a **do** clause, two copies of the expression and generator frame pointers must be saved, one for **break** and one for **next**. At the beginning of the **do** clause, the copy used by **next** is updated so that the generators in the **every** clause are reactivated at the proper points.

3.3 Storage Allocation and Reclamation

During program execution, storage allocation is necessary when a data object is created. The two primitive routines *allocate* and *alcstr* allocate storage in the heap and string space, respectively. Both routines return pointers to the beginning of the newly allocated regions. Neither routine 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 heap, *allocate(n)* returns a pointer to *n* contiguous bytes of storage. Because a wide variety of objects may reside in the heap, 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 heap. 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, and establish the block header for that datatype within the newly allocated region.

In the string space, *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.

Source code for all of the allocation routines is contained in the file *rt/alc.c*. Almost all interaction with the storage management is made through these routines. Two exceptions occur in string concatenation and reading a fixed number of bytes. In each case, it is simpler and more efficient to have these operations deal directly with storage management.

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 up 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. As an aid, space for several descriptors is automatically protected by the procedure invocation mechanism, and is usually used to hold information pertaining to the arguments of the procedure (see Section 3.4).

Routines to ensure space are provided for each of the two storage regions. The routine *sneed(n)* ensures that at least *n* bytes of storage remain in the string space, and *hneed(n)* performs the same function in the heap. If either routine finds that there is insufficient storage remaining, it will invoke the *garbage collector* in an attempt to obtain that storage. If that fails, then program execution is aborted with an appropriate diagnostic.

Garbage collection, or *storage reclamation*, is a process that identifies all valid data in storage and compacts that data in order to provide a contiguous area of unused storage. The algorithm used for identifying valid data is based upon the algorithm described by Hanson [6]. Only the more novel features are discussed here.

Whenever a predictive need request discovers that insufficient storage remains in either the heap or string space, the garbage collector is invoked to free up space in both regions. This approach is more efficient in situations where both regions are heavily allocated, and only slightly less efficient otherwise.

The approach is to sweep through the permanent data regions and the stack, looking for descriptors that are either pointers into the heap or string qualifiers. When a string qualifier is found, a pointer to that qualifier is saved in a temporary data region at the end of the heap. If the descriptor is a pointer into the heap, then that heap data 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 heap

blocks, all data 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 need only know the size of the block and the location of the first descriptor. Information concerning a data block's size, as well as the offset for the first descriptor is maintained in the file *rt/dblocks.c*.

After the marking phase is completed, the string region is compacted. The algorithm used is described by Hanson [8]. The pointers to the string qualifiers are sorted so that the order of all valid strings within the string space is identified. The string qualifiers are then processed in order, and modified as the valid strings are compacted. If this compaction does not free up enough space within the string space to satisfy the request, the heap must be moved in order to provide more room in the string space. An attempt is also made to provide some additional "breathing room" in the string space to permit future expansion.

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

This method has proved to be quite satisfactory for most applications. 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 string qualifiers, even though enough storage would become available if the heap were collected separately. In practice, this has not been a problem. The source code for the garbage collector is maintained in the file *rt/gc.c*.

3.4 Coding Conventions

The use of *usignal* and *save* to implement generators and the backtracking mechanism has been discussed, as well as accessing arguments and the predictive need allocation scheme. Several other aspects of the runtime system require more detailed explanations.

There are several linked files in the runtime system, providing some communication between the various portions of the Icon system. Two sets of linked files bear special notice. The file *lib/lib.h* is linked to *rt/rt.h*, and *lib/keyword.h* is linked to the translator file *tran/keyword.h*.

The file *lib/lib.h*, (or its link *rt/rt.h*) is included by every source file in the runtime system, and contains machine-dependent defined constants, runtime data structure declarations, external declarations and defined constants and macros for flags, type codes, argument accessing, and bit manipulations.

The macros *tsth* and *seth* are the basic primitives used in conversions between csets and strings. These are defined as macros rather than as procedures for efficiency: both appear within tight loops where the overhead for calling procedures would be a significant portion of the processing time. However, because the arguments appear several times within the macro expansion, care must be taken to avoid auto-incrementing the arguments.

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 procedure. Any routine calling for type conversion must determine if heap or string space storage is needed, 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* or *long integer*, and makes no attempt to distinguish between short and long integers.

As mentioned in Section 3.3, there is space set aside to hold temporary descriptors and to protect the validity of these descriptors during garbage collection. The garbage collector knows about this region, and *tends* it during storage reclamation. The region is defined in the file *rt/start.s*, and is bounded by the labels *tended* and *etended*. To simplify access while programming in C, several global variables point into this tended region. The most commonly used is the array of descriptors *arg*, which is used to hold the dereferenced values of arguments to functions and operators. Examination of any of the function or operator procedures shows

the use of *arg*. There is room for six descriptors in *arg*, since that is the maximum number of arguments to any function or operator. Since a garbage collection can occur only during a call to *sneed* or *hneed*, or between suspension and reactivation, the only places where C routines need to ensure that all pointers into the heap or string space are tended are just before calls to *sneed*, *hneed*, or *save*.

In addition to *arg*, there are several other variables in the tended region. To keep the size of the tended region small, these have been equivalenced to *arg*. The equivalences are defined in *rt/start.s*.

All Icon procedure and function names are folded by the translator to lower-case with a capitalized first letter. This prevents name collisions between Icon procedures and other routines, such as those for operators, type conversions, and storage management. It is also the only remaining motivation for the equivalence of upper- and lower-case letters in Icon source programs.

4. Modifying the Implementation

This section is intended to serve as a brief guide for those who wish to modify the Icon system. It is not comprehensive; it only points to various parts of the implementation that need to be considered when making various kinds of changes.

Perhaps the most common kind of change that one might expect to make is to add new functions (built-in procedures). To add a function, first write it according to the conventions described in Section 3.4. (Use an existing function similar to the new one as a prototype.) Be especially careful to observe the rules concerning storage allocation and tended descriptors. Then prepare to add the new function to the runtime library by moving the source code into the *lib* directory and adding its name to *lib/Makefile* (the name must be added in three places — there are many examples already in the makefile). Then add the name to the file *link/builtin.c* in proper alphabetical order for use by the linker.

The makefile *bin/Makefile* is set up to compile whatever needs to be compiled to make a new system. When all changes have been made to the source code, simply change to the *bin* directory and run *make*. This runs *make* in each of the four system directories — *tran*, *link*, *lib*, and *rt* — and then copies the new versions into the *bin* directory.

Adding new operators is more complicated — this is described in detail since many other kinds of modifications require many similar changes. Again, the first step is to write the routine, place it in the *lib* directory, and add its name to the makefile. Next, the operator must be added to the translator, as follows:

(1) Add the operator to the operator table in *tran/optab*; the structure of the table is described in Section 1.1.

(2) Create a unique name for the new token and make a new token table entry in *tran/toktab* in the operators section of the table. Although the operators section of the table is in alphabetical order by token name as distributed, there is no need to preserve this order. (Do not put any tabs in the file *toktab* if it is to be processed by the SNOBOL4 program discussed in Section 1.1.)

(3) If SNOBOL4 is not available, edit the files *tran/optab.c* and *tran/toktab.c* to correspond to the changes made in steps 1 and 2. This sometimes involves a renumbering of token table entries in both files (but nowhere else).

(4) Add the operator to the grammar in *tran/ucon.g*. The token name must be added to the list of terminal symbols at the beginning of the grammar file, and the operator must be inserted into the syntax at the appropriate precedence level. If the precedence is the same as that of an existing operator, simply add the operator as an alternative to the existing production; otherwise, insert a new production, and change the production at the next lower precedence level to refer to the new one. The semantic action should create either a *BINOP* or a *UNOP* node in the parse tree; use existing actions as a prototype.

(5) The new operator must now be added to the code generator in *tran/code.c*. Insert a case in either of the routines *binop* or *unop* for the new token name that assigns a new intermediate code opcode to *name*, as for other operators — this causes the new opcode to be emitted into the *ucode*. The opcode should have the same name as the library routine that performs the operation. If the evaluation of the operator can fail, then increment the variable *fail* — this causes a failure test to be emitted after the operator.

The new intermediate code opcode must also be added to the linker. Add a defined constant to *link/opcode.h*; order here is not important. Then add the opcode name and the defined constant to *link/opcode.c*;

alphabetical order must be preserved, since a binary search is used. Then edit the code generator in *link/code.c*, adding a case in the routine *gencode* with either the binary or the unary operators. The standard processing here emits code that evaluates the operand(s), then calls a library routine with the same name as the intermediate code opcode. The system is then be ready to be made as described above.

Adding a new control structure is similar in nature to adding a new operator. Most often, a new reserved word must be added to *tran/tokens*; this part of the token table must be kept in alphabetical order. The new token must be added to the grammar, and productions must be added, usually at the highest precedence level (the same as **if**, for example). The semantic action for the new production will probably involve creating a parse tree node of a new type. The new node type should be added to *tran/tree.h* and a new case in the routine *traverse* (in *tran/code.c*) should be added to generate intermediate code. The intermediate code generated can use any of the existing opcodes or can use new ones created specifically for the new control structure. If new opcodes are created, they must be added to the linker as described above, and a new case in the routine *gencode* must generate code for it. The generated code can be either entirely in-line or can call a new library routine (see, for example, the generated code for **scan** expressions). If new code generation templates are needed, modify the routine *emit* in *link/code.c* and the list of templates in *link/code.h*. If the code calls a new library routine, add it to *lib* as described above, then the system is ready to be made.

Modifying the semantics of existing control structures, operators, or functions, often involves changing only the generated in-line code or a library routine. Modifying the syntax without disturbing any semantics usually requires only a change to the grammar.

Adding a new datatype means making many of the above changes. A new datatype code must be added to *lib/lib.h*, and a new data block format must be defined, if necessary. The size and location of the first descriptor of the new data block must be entered in *rt/dblocks.c* so that the garbage collector knows how to treat the block. The routines in *lib/image.c* and *rt/outimage.c* must be extended so that images of the new datatype can be produced. New functions and operators need to be created, and possibly new coercion routines must be added to *rt*.

Adding a new keyword entails a change to *tran/keywords* (and, if SNOBOL4 is not available, to *tran/keywords.h*) and a new case in *lib/keywd.c*. The file *lib/keywords.h* is a link to *tran/keywords.h*, so the two are modified simultaneously. Many keywords require trapped variables, which requires changes to *lib/lib.h*, *lib/asgn.c*, and *rt/deref.c*; the trapped variable for *&subject* serves as a good model.

As mentioned above, the examples in this section are intended to identify what parts of the system are affected by certain kinds of changes or extensions. A thorough understanding of the system is suggested, however, for other than minor changes.

Acknowledgements

Many features of the current implementation of Icon are based upon the original Ratfor implementation by Dave Hanson, Tim Korb, and Walt Hansen [2, 9]. We would like to thank Ralph Griswold and Dave Hanson for their many suggestions regarding the implementation and for many careful readings of this paper.

References

[1] Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language, Version 3.* Technical Report TR 80-2, Department of Computer Science, The University of Arizona, Tucson, Arizona, May 1980.

[2] 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.

[3] Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[4] Johnson, Stephen C. "Yacc: Yet Another Compiler-Compiler." *Unix Programmer's Manual, Seventh Edition.* Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, 1979.

[5] Aho, A. V., and S. C. Johnson. "LR Parsing." *Computing Surveys* 6, 2 (June 1974), 99-124.

[6] Hanson, David R. "Storage Management for an Implementation of SNOBOL4." *Software—Practice and Experience* 7, 2 (March 1977), 179-192.

[7] Hanson, David R. "Variable Associations in SNOBOL4." *Software—Practice and Experience* 6, 2 (April 1976), 245-254.

[8] 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.

[9] 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.

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 central 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.

Six of the nodes — **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.

alt An alternation (the | operator).
- *tree0* The left operand.
- *tree1* The right operand.

and A conjunction (the & operator).
- *tree0* The left operand.
- *tree1* The right operand.

augop An augmented assignment.
- *tree0* The operator (pointer to an **op** node).
- *tree1* The left operand.
- *tree2* The right operand.

binop A binary operation.
- *tree0* The operator.
- *tree1* The left operand.
- *tree2* The right operand.

break A **break** expression.

case A **case** expression.
- *tree0* The control expression.
- *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, as in *e1*: *e2*.
- *tree0* The case selector expression, *e1*. For a **default** clause, this field points to a **res** node that contains the reserved word **default**.
- *tree1* The expression, *e2*, that is executed if the selector matches the control expression.

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.

drive A driven expression.
- *tree0* The driven expression.

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 (the . operator)
- *tree0* The left operand.
- *tree1* Pointer to an **id** node, containing the field name.

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

if An **if** expression.
- *tree0* The control expression.
- *tree1* The **then** clause.
- *tree2* The **else** clause.

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

invok A procedure call (invocation).
- *tree0* The expression naming the procedure.
- *tree1* The argument list. 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.
- *val0* The index of the referenced keyword, defined in the file *tran/keyword.h*.

list A list construction, as *[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.
- *tree0* The style of loop. This field points to a **res** node, which identifies the reserved word that introduced the loop.
- *tree1* The control expression.
- *tree2* The **do** clause.

next A **next** expression.

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.

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 **scan** or **transform** expression.
- *tree0* The reserved word **scan** or **transform**.
- *tree1* The control (scanned) expression.
- *tree2* The **using** clause.

sect A section operation, as *e1*[*e2*:*e3*].
- *tree0* The first operand, *e1*.
- *tree1* The second operand, *e2*.
- *tree2* The third operand, *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.
- *tree0* The expression following the **suspend**.

toby A **to-by** operation.
- *tree0* The initial value expression.
- *tree1* The **to** clause.
- *tree2* The **by** clause.

to A **to** operation.
- *tree0* The left operand.
- *tree1* The right operand.

unop A unary operation.
- *tree0* The operator.
- *tree1* The operand.

Appendix B

Icon Formal Syntax

The following grammar describes the Icon language. Reserved words are shown in boldface; all operators are shown in Roman. The non-terminals *ident*, *literal*, and *empty* are left undefined in the syntax.

program → *decls*

decls → *empty*
→ *decls decl*

decl → *record*
→ *proc*
→ *global*

global → **global** *idlist*

record → **record** *ident* (*arglist*)

proc → *prochead* ; *locals initial nprocbody* **end**

prochead → **procedure** *ident* (*arglist*)

arglist → *empty*
→ *idlist*

idlist → *ident*
→ *idlist* , *ident*

locals → *empty*
→ *locals retention idlist* ;

retention → **local**
→ **static**
→ **dynamic**

initial → *empty*
→ **initial** *dexpr* ;

nprocbody → *empty*
→ *procbody* ;

procbody → *ndexpr*
→ *procbody* ; *ndexpr*

ndexpr → *empty*
→ *dexpr*

dexpr → *expr*

nexpr → *empty*
→ *expr*

expr → *expr1*
→ *expr* & *expr1*

expr1 → *expr2*
→ *expr2 op1 expr1*
→ *expr2 op1a expr1*

op1 → := | :=: | <- | <-> | ::=

op1a → +:= | -:= | *:= | /:= | %:= | ↑:= | ++:= | --:= | **:= | ||:=

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

expr3 → *expr4*
→ *expr3* | *expr4*

expr4 → *expr5*
→ *expr4 op4 expr5*

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

expr5 → *expr6*
→ *expr5* || *expr6*

expr6 → *expr7*
→ *expr6 op6 expr7*

op6 → + | - | ++ | --

expr7 → *expr8*
→ *expr7 op7 expr8*

op7 → * | / | % | **

expr8 → *expr9*
→ *expr9* ↑ *expr8*

expr9 → *expr10*
→ *expr9* **fails**

expr10 → *expr11*
→ *op10 expr10*

op10 → | | . | ! | + | - | ~ | =

expr11 → *ident*
→ *literal*
→ & *ident*
→ *expr11* . *ident*
→ *expr11* [*expr*]
→ *expr11* (*exprlist*)
→ [*exprlist*]
→ (*expr*)
→ { *procbody* }
→ *while*
→ *until*
→ *every*
→ *repeat*
→ **next**
→ **break**
→ *if*
→ *case*
→ *scan*
→ *return*
→ *section*

while → **while** *dexpr*
→ **while** *dexpr* **do** *dexpr*

until → **until** *dexpr*
→ **until** *dexpr* **do** *dexpr*

every → **every** *expr*
→ **every** *expr* **do** *dexpr*

repeat → **repeat** *dexpr*

if → **if** *dexpr* **then** *dexpr*
→ **if** *dexpr* **then** *dexpr* **else** *dexpr*

case → **case** *dexpr* **of** { *caselist* }

caselist → *cclause*
→ *caselist* ; *cclause*

cclause → **default** : *dexpr*
→ *expr* : *dexpr*

scan → **scan** *dexpr* **using** *dexpr*
→ **transform** *dexpr* **using** *dexpr*

return → **fail**
→ **return** *ndexpr*
→ **suspend** *nexpr*

section → *expr11* [*expr sectop expr*]

sectop → : | +: | −:

exprlist → *nexpr*
→ *exprlist* , *nexpr*

Appendix C

The Intermediate Language

The intermediate language generated by the Icon translator, *ucode*, resembles a stack-oriented assembler 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 driven expressions; generator frame markers, which mark dormant generators; and loop frame markers, which mark the stack for loop exits. 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 format similar to the ucode file; 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 runtime actions for which code must be generated.

bevery
Save the contents of *lptop* on the stack and create a new loop frame for an **every-do** loop.

bloop
Save the contents of *lptop* on the stack and create a new loop frame for an **every** (without a **do** clause), **while**, or **until** loop.

bscan
Save the scanning subject and position on the stack, and establish a new subject and position.

ccase
Duplicate the value on the top of the stack after creating a new expression frame. Used in **case** expressions.

drive
Drive the current expression, as delimited by this opcode and the matching **mark** above it, to success. That is, if the signal is failure and alternatives exist, reactivate the most recent dormant generator. Otherwise, exit the current expression frame.

dup
Duplicate the value on the top of the stack. Used in augmented assignments.

eevery
Restore the value of *lptop* from the stack and exit the current loop frame. The null value is pushed onto the stack, and the signal is set to success.

eloop
Same as **eevery**.

escan
Restore *&subject* and *&pos* from the stack.

every
Update the second copy of the stack heights saved in the current **every-do** loop frame. This is done between an **every** clause and its **do** clause, so that **next** statements work properly.

evnext *lab*
Reset the expression and generator frame pointers from the second copy of the stack heights saved in the current **every-do** loop frame, then go to *lab*, which is the **next** label for the current loop.

field *name*
Access the field *name* from the record object 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 and after every procedure call.

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 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 or record creation object is on the stack, just beyond the arguments. After invocation, the arguments are popped from the stack, and the returned value is pushed (see **return**).

invsig
Invert the signal. If it was success, make it failure; if it was failure, make it success and replace the top of the stack with the null value.

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

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

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

lpnext *lab*
Reset the control stack pointer and the stack marker from the current loop frame, then go to *lab*, which is the **next** label for the current loop.

mark
Save the current expression and generator frame pointers on the stack, then create a new expression frame. This opcode and the matching **drive** below it define the boundaries of a driven expression.

pnull
Push the null value onto the stack.

pop
Pop the top element off of the stack.

pushl
Push the integer 1 onto the stack.

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

return
Return from an Icon procedure (see **invoke**). The value on the top of the stack is saved, the stack is restored to the previous procedure frame, and the saved value is pushed onto the stack as the returned value of the returning procedure.

save *lab*
Create a new generator frame, so that a generator may be reactivated at *lab*. This opcode effectively creates a "dormant" generator; it is used for alternation, so that the second alternative may be activated when needed. The other generators, except for **suspend**, are part of the runtime library, and create a new control stack frame implicitly.

sig=0
Set the signal to failure.

sig=0? *lab*
If the signal is failure, go to *lab*.

sig=1
Set the signal to success.

sig=1? *lab*
If the signal is success, go to *lab*.

str *n*
Push the string literal at constant table location *n* onto the stack.

susp
Suspend the current procedure. This opcode creates a new control stack frame, so that the suspending procedure becomes a dormant generator, then returns.

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

xform
Assign the value of *&subject* to the variable being scanned, then restore the previous *&subject* and *&pos* from 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 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, the signal is set to failure.

abs	$\|x$	**numgt**	$x > y$
asgn	$x := y$	**numle**	$x <= y$
bang	$!x$	**numlt**	$x < y$
cat	$x \mid\mid y$	**numne**	$x \sim= y$
compl	$\sim x$	**plus**	$x + y$
diff	$x -- y$	**power**	$x \uparrow y$
div	x / y	**rasgn**	$x <- y$
eqv	$x === y$	**rswap**	$x <-> y$
inter	$x ** y$	**sasgn**	$x ::= y$
minus	$x - y$	**sect**	$x[y:z]$
mod	$x \% y$	**strne**	$x \sim== y$
mult	$x * y$	**streq**	$x == y$
neg	$-x$	**subsc**	$x[y]$
neqv	$x \sim=== y$	**swap**	$x :=: y$
numeq	$x = y$	**tabmat**	$=x$
numeric	$+x$	**toby**	x **to** y **by** z
numge	$x >= y$	**unionc s**	$x ++ y$

Non-Imperative Instructions

The following instructions generate no executable code. Instead, they communicate various information to the linker about the 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 **locend** instruction, a sequence of **con** instructions, 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 its scope, retention, and other information. 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.

locend

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

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 literals, it is a comma-separated list of 3-digit octal numbers, each representing one byte in the string.

end

Signal the end of a procedure.

Global Symbol Table Instructions

A single 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** switch was given on the translator command line, otherwise it is **local**.

trace

Enable runtime tracing. This instruction is present if the **-t** switch 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).

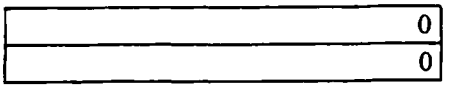
Appendix D

Data Representations

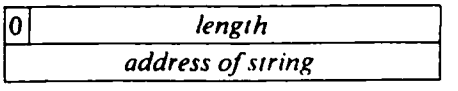
Descriptor Formats

The figures below depict each of the six descriptor types mentioned in Section 3.1. Each descriptor is two 16-bit words long; the first word is shown on top of the second.

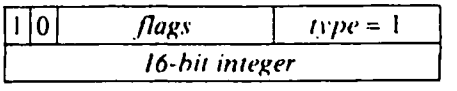
Null



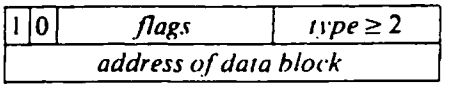
String Qualifier



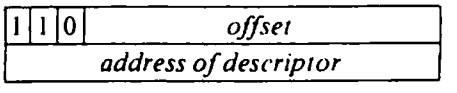
Short Integer



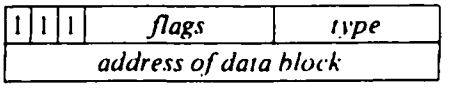
Value



Variable



Trapped Variable

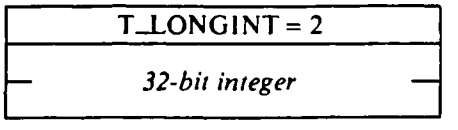


Data Block Formats

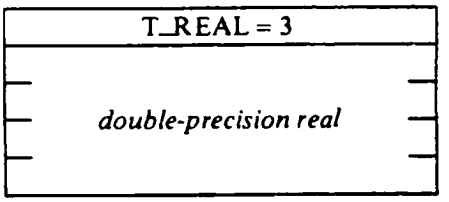
The data blocks used by the Icon system are pictured below. The data type code, shown as both a mnemonic and an integer, is always the first word of the block and has the same value as the type code in the *value* or *trapped variable* descriptor that refers to it. All *name* fields in the data blocks are *string qualifier* descriptors, and all *pointers* in the data blocks are *variable* descriptors.

Variable-length blocks and especially long blocks are shown with a break in the side border.

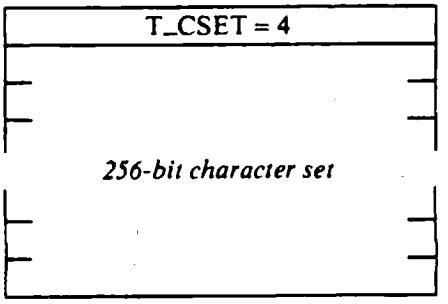
Long Integer



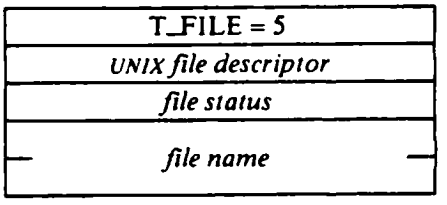
Real



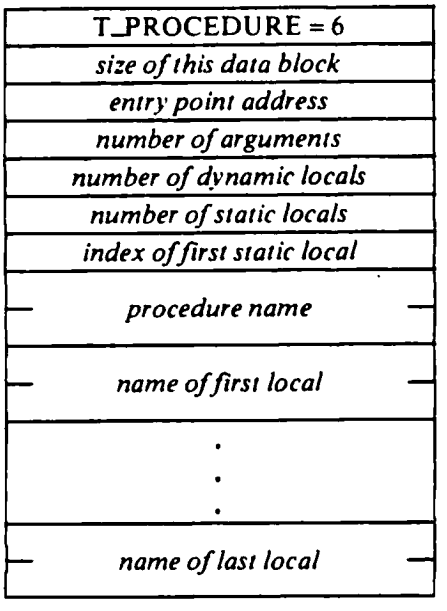
Cset



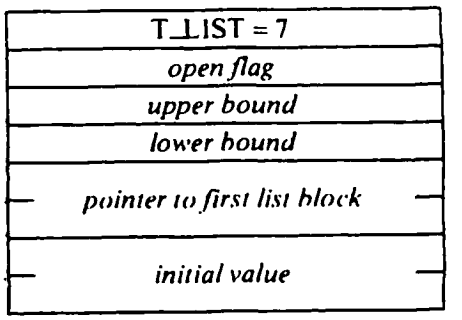
File



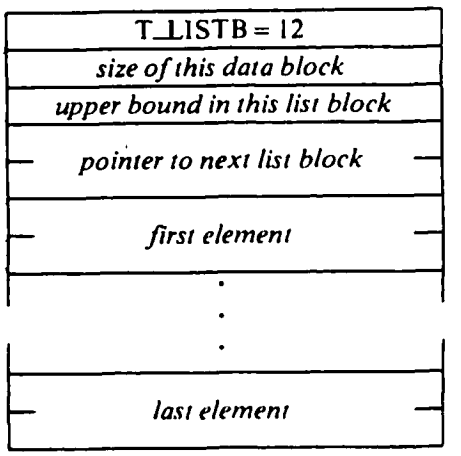
Procedure



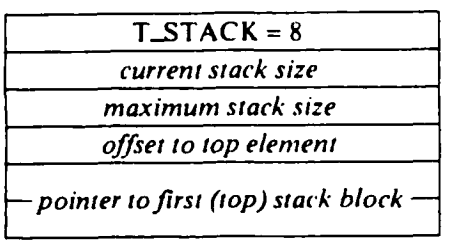
List



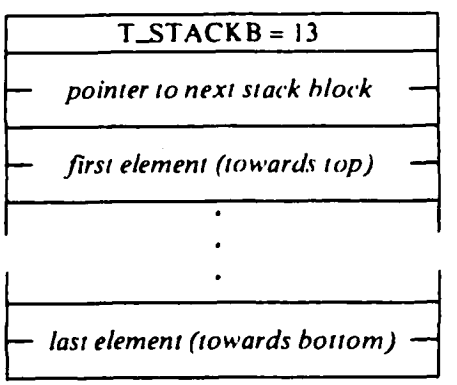
List Block



Stack



Stack Block



Table

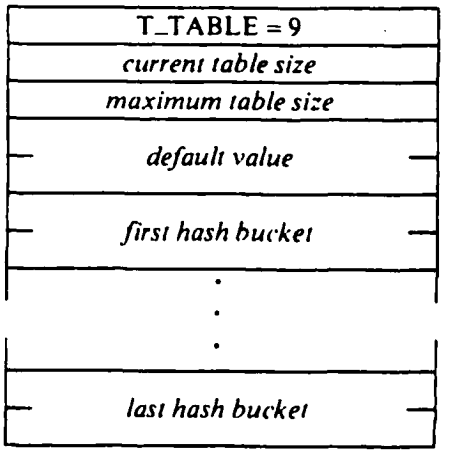
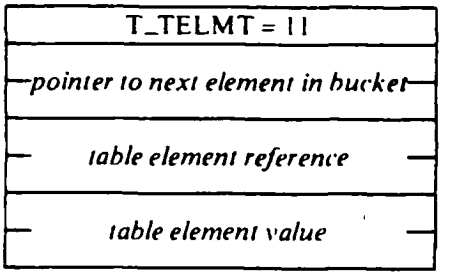
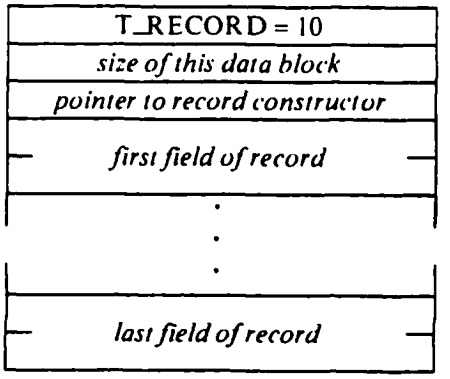


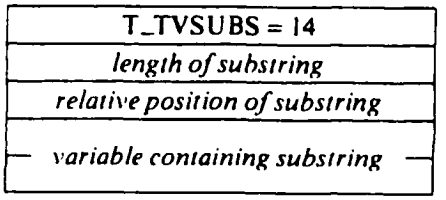
Table Element



Record



Substring Trapped Variable



Subject Trapped Variable

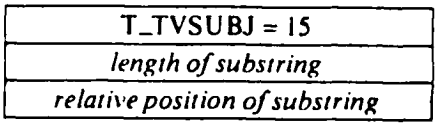
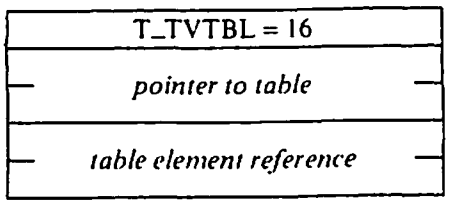
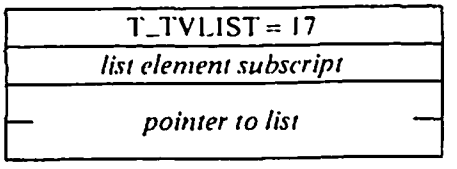


Table Element Trapped Variable



List Element Trapped Variable



Appendix E

Stack Frame Formats

The four kinds of stack frames are described below. For each kind of frame, a *frame pointer* points to the most recent *frame marker*, which marks one end of the frame. Each frame marker contains a pointer to the next most recent marker of the same kind.

On the PDP-11, the frame pointers are contained in registers *r2-r5* whenever an Icon procedure is active. The procedure frame pointer is in *r5*, the expression frame pointer is in *r4*, the generator frame pointer is in *r3*, and the loop frame pointer is in *r2*. When a C procedure is active, only the procedure frame pointer is kept in a register; registers *r2-r4* are used for local variables by C procedures.

Procedure Frames

A procedure frame contains a procedure's arguments, local variables, and temporary storage for incomplete computations. When an active procedure invokes another procedure, a new procedure frame is created for the new procedure, which then becomes active. As such, the new procedure represents an incomplete computation in the calling procedure, so the new procedure frame is "nested" within the old one. The *procedure marker* is placed on the stack between the arguments and local variables. The format of the procedure marker is shown in the following table; the locations are shown relative to the contents of *r5*, the procedure frame pointer.

-8(r5)	*previous source program line number*
-6(r5)	*previous contents of r2 (loop frame pointer)*
-4(r5)	*previous contents of r3 (generator frame pointer)*
-2(r5)	*previous contents of r4 (expression frame pointer)*
0(r5)	*previous contents of r5 (procedure frame pointer)*
2(r5)	*return address*
4(r5)	*number of arguments*

Expression, generator, and loop frames are always contained wholly within a procedure frame, and their respective frame pointers are cleared to zero after being saved in the procedure marker.

The first argument to a procedure is located at 6(r5), the second at 10(r5), and so on. The first local variable is located at -12(r5), the second at -16(r5), and so on.

Procedure markers created for functions and operators do not contain the source program line number, since functions and operators do not change it. Because they are C procedures, their local variables are not descriptors and are subject to C language conventions, but everything above the marker (higher addresses) is subject to Icon language conventions. The location of the procedure marker for functions and operators is considered the *boundary*, mentioned in Section 3.2.

Expression Frames

An expression frame limits the scope of backtracking. No dormant generator outside the current expression frame may be reactivated until evaluation of the current expression is complete. The format of an expression marker is shown in the following table; locations are shown relative to *r4*, the expression frame pointer.

-2(r4)	*previous contents of r3 (generator frame pointer)*
0(r4)	*previous contents of r4 (expression frame pointer)*

When an expression frame is created, the generator frame pointer is cleared after being saved in the expression marker, to indicate that there are no dormant generators that may be reactivated while the new expression frame is current. An expression frame extends from its expression marker to the top of the stack. Expression frames are not disjoint; new frames are always nested within older ones.

Generator Frames

A generator frame preserves the state of execution of a dormant generator. When a suspending procedure calls *save*, a generator marker is placed on the stack to mark the point of suspension, then the most recent expression frame *outside* the suspending procedure frame (the expression frame that was current just prior to invocation of the suspending procedure) is then duplicated and pushed onto the stack. The suspending procedure then returns, so that the expression frame that was duplicated is current. Thus, the generator frame is contained within the expression frame, and "hides" the dormant generator. The format of the generator marker is shown in the following table; locations are shown relative to *r3*, the generator frame pointer.

-2(r3)	*previous value of &level*
0(r3)	*previous boundary address*
2(r3)	*previous contents of r2 (loop frame pointer)*
4(r3)	*previous contents of r3 (generator frame pointer)*
6(r3)	*previous contents of r4 (expression frame pointer)*
8(r3)	*previous contents of r5 (procedure frame pointer)*
10(r3)	*reactivation address*

The last five words of the generator marker are actually part of a procedure marker, created by the call to *save*. Thus, the reactivation address is just the return address for *save*.

When a function or operator suspends, there is a boundary that becomes hidden. This boundary address needs to be restored upon reactivation. It is also important to the garbage collector, since the portion of a generator frame between the hidden boundary and the generator marker does not have the well-defined structure required.

Loop Frames

A loop frame is created for loops that can be affected by **break** or **next**. A loop marker is pushed onto the stack when a loop is entered; it records the expression, generator, and loop frame pointers at the time the loop was entered. For **every-do** loops, two copies of the expression and generator frame pointers are saved. The format of the loop marker is shown in the following table; locations are shown relative to *r2*, the loop frame pointer.

-4(r2)	*previous contents of r3 (generator frame pointer)*
-2(r2)	*previous contents of r4 (expression frame pointer)*
0(r2)	*previous contents of r2 (loop frame pointer)*
2(r2)	*previous contents of r3 (generator frame pointer)*
4(r2)	*previous contents of r4 (expression frame pointer)*

The first two words are present only for **every-do** loops, and are actually just an expression marker. These copies of the expression and generator frame pointers are updated each time the **do** clause begins, and are used to restore the state of the control expression upon execution of a **next**.