The Run-Time Implementation Language for Icon V8.7*

Kenneth Walker

TR 92-18

August 1, 1992

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants CCR-8901573 and DCR-8502015.

The Run-Time Implementation Language for Icon V8.7

1. Introduction

This report describes a language to aid in the implementation of a run-time system for the Icon language [1]. This language is called RTL. The design of RTL is motivated by the needs of the optimizing compiler for Icon, iconc. The optimizing compiler needs a variety of information about run-time routines and needs to be able to use both general and specialized versions of these routines to generate efficient code.

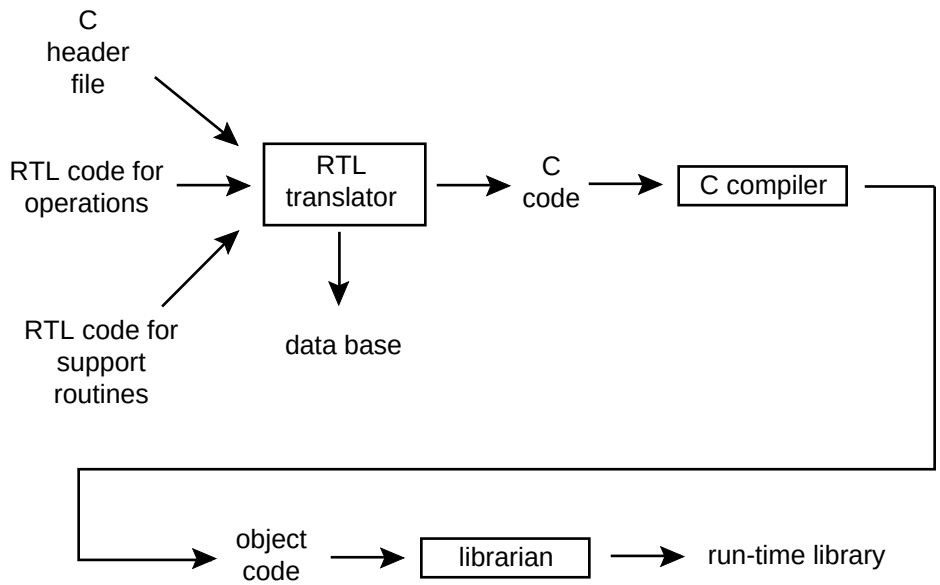
RTL is used to define the operators, built-in functions, and keywords of the Icon language. It consists of two sub-languages: an interface and type-specification language, and a slightly extended version of C. The interface language defines, in part, what an operation looks like within the Icon language (some operators have a non-standard syntax that is determined by iconc's parser, but the operators are specified in the interface language the same way as operators with a standard syntax). The interface language also specifies the type checking and conversions needed for arguments to the operation. Several conversions to C values are included for convenience and efficiency. The interface language presents the overall structure of the operation's implementation. The extended C language is embedded within certain constructs of the interface language and provides the low-level details of the implementation. The extensions include operations for manipulating, constructing, and returning Icon values.

RTL includes features designed to support optimizations that have not yet been implemented in iconc. This report describes how the features should be used to provide support for those future optimizations.

When referring to files in the Icon implementation, this report gives path names relative to the Icon source directory, src. For example, if Icon is installed in /usr/icon/v8 then the file runtime/Makefile has a full path name of /usr/icon/v8/src/runtime/Makefile.

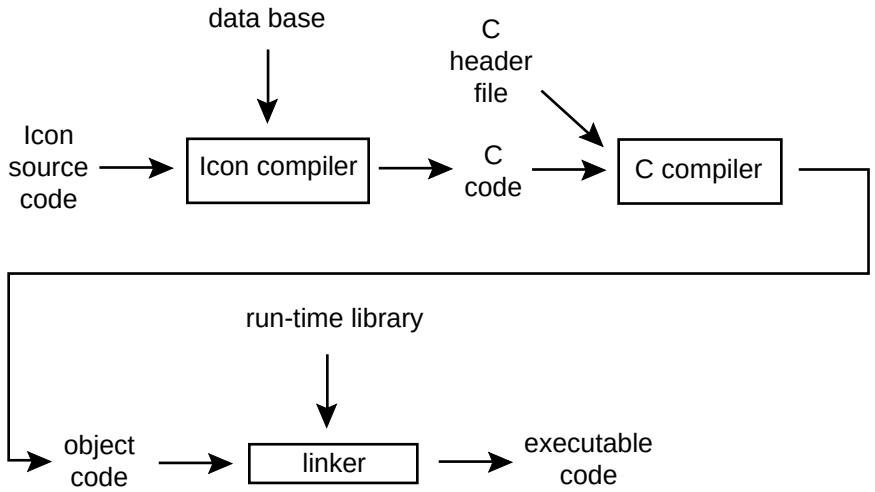
2. Overview of the Icon Interpreter/Compiler System

The translator for RTL is called rtt. By default, it produces C code for use with iconc. This C code is compiled and placed in an object code library. The translator also creates a data base of information about Icon operations:

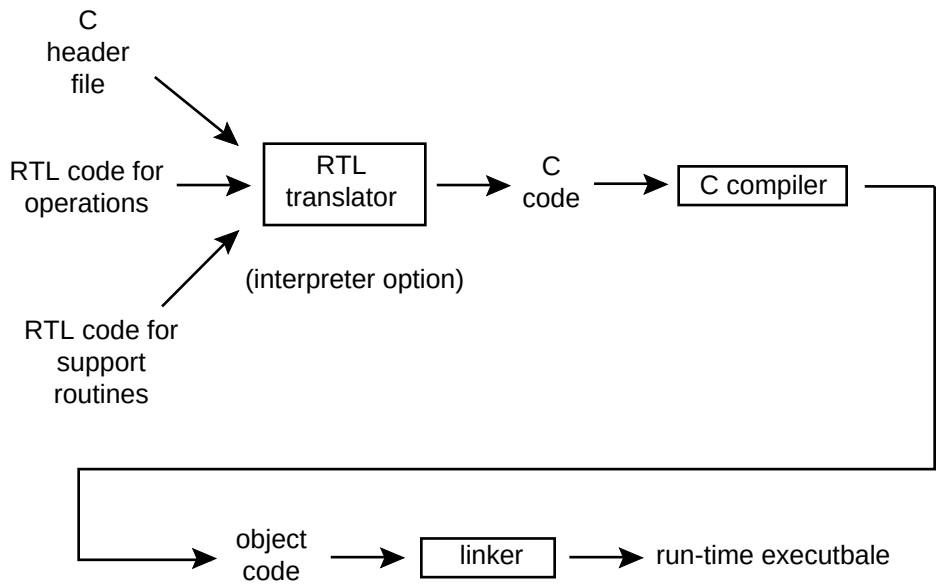


The Icon compiler produces an executable program from Icon source code by translating Icon procedures into C

functions, supplying a C main function, and compiling and linking these functions with the library produced using rtt. The compiler makes use of the data base when performing the translation:



rtt has an option to produce C code for building an Icon interpreter. Most of the input is identical to that used for the compiler, but additionally it includes code for the interpreter proper. The resulting C code is compiled and linked to produce the executable interpreter, iconx. No data base is created with this option:



For the interpreter, Icon programs are translated into an intermediate form, which is then interpreted by iconx. iconx's interpreter loop calls the operations and support routines in the run-time system as needed.

RTL files have a suffix of .r. The names of the output files vary depending on whether an interpreter or compiler library is created; this is described below. The output files are given the suffix expected by the C compiler; this is typically .c.

2.1 Preprocessor

rtt has a built-in C preprocessor based on the ANSI C Standard [2], but with extensions to support multi-line macros with embedded preprocessor directives [3]. rtt automatically defines the identifier StandardPP before processing any files to indicate that it is an ANSI C Standard preprocessor; some parts of the Icon system, for example header files, may also be used directly with non-ANSI C compilers and contain code conditional on this identifier.

rtt processes a standard include file, grttin.h, before every RTL file it translates; grttin.h contains #include directives for several other files. rtt uses these files to establish macro definitions and to determine typedef names, but it outputs nothing while processing these files.

Within its own preprocessor, rtt defines COMPILER to 0 or 1 depending on whether interpreter or compiler code is being generated. It also places a #define directive for COMPILER at the beginning of every output file. It produces an #include directive for rt.h after the #define directive. The preprocessor directives in rtt's output are used by the C compiler.

Arbitrary text may be passed through rtt using the #passthru directive. The contents of a #passthru directive are written to the output after each sequence of white space within the contents is replaced by a single space character. #passthru is useful in those rare circumstances where something needed by the C compiler must be hidden from rtt. #passthru only works correctly at the global level; if it is used within functions or declarations, the result is not put in the correct location in the output. For example

```
#passthru #define LIB_GET_EF      LIB$GET_EF
```

in the input results in

```
#define LIB_GET_EF LIB$GET_EF
```

in the output.

Such code is used under VMS to allow calls to a function with a non-standard identifier for a name. RTL code for VMS contains calls to LIB_GET_EF. rtt parses this name as an identifier (it produces an error message if it tries to parse LIB$GET_EF) and copies it to the output. The preprocessor for the C compiler sees the #define directive and changes instances of LIB_GET_EF to the real function name, LIB$GET_EF, which is what the VMS C compiler needs.

2.2 Support Routines

The *primary* run-time routines written in RTL can call support routines written in C. The support routines may either be written in standard C, such as those in the file common/time.c, or they may use some of the features of extended C. In the former case, they may be compiled directly with the C compiler. It does no harm to put standard C code through rtt, but it is important to determine which header files are automatically included by rtt so they are not explicitly included in the C code.

2.3 The Data Base

rtt uses the data base to pass various information about run-time routines to iconc, including the names of the library routines that implement various operations. iconc's compile-time analyses, including type inference, provide information that the code generator needs to produce good code. These analyses need information about Icon's operations. The control characteristics of an operation (that is whether it suspends, fails, etc) are explicitly stated in the operation's code. Type inference needs additional information; some of this is implicitly supplied through the type-checking code and some is explicitly supplied through *abstract type computations*. This information is stored in the data base.

iconc uses the data base to determine what built-in functions and keywords are available. Therefore new functions and keywords can be added to the the compiler without rebuilding iconc. They need only be run through rtt, updating the data base and the library. (See [4] for instructions on adding new built-in functions to the interpreter.)

Externally, the data base is implemented as a text file. It is organized to be read by programs, but can be read by humans beings without great difficulty. See Appendix E for the data base format.

2.4 Versions of an Operation

There is one RTL source coding for an Icon operation. However, each operation has two implementations in the compiler: an in-line version and a most general version (except keywords, which only have an in-line version). The most general version is a C function in the library. This function conforms to a standard calling convention and includes code to perform argument list adjustment and dereferencing, along with type checking and conversions. iconc's code generator uses this version of the operation when it is impossible or inappropriate to use the in-line version.

The in-line version of an operation is stored by rtt in the data base. The entire interface portion of an operation, including type checking and conversions, is represented in the data base. However, the detailed code may either be put in the data base or placed in the library as C functions. This choice is controlled by the programmer who writes the run-time routine. One operation may produce several of these detail functions (typically different functions for different argument types). They have calling conventions specific to their needs. The calling conventions are noted in the data base at the point in the in-line code where the functions must be called.

The interpreter has one version of each operation. It is similar to the most general version for the compiler, but the calling conventions differ.

2.5 Operation Documentation

RTL allows explicit documentation of an operation. In addition, many of the characteristics of an operation that are used by the compiler can be automatically added to the description of the operation. Thus, the implementation acts as a central location for operation documentation. This documentation can be extracted from the data base and formatted for various purposes.

3. Language Definition

The grammar for RTL is presented in extended BNF notation. Terminal symbols are set in Helvetica. Non-terminals and meta-symbols are set in *Times-Italic*. In addition to the usual meta-symbols, *::=* for ''is defined as'' and | for ''alternatives'', brackets around a sequence of symbols indicate that the sequence is optional, braces around a sequence of symbols followed by an asterisk indicate that the sequence may be repeated zero or more times, and braces followed by a plus indicate that the enclosed sequence may be repeated one or more times.

An rtt input file may contain operation definitions written in RTL, along with C definitions and declarations. This section describes the operation definitions. C language documentation should be consulted for ordinary C grammar. The next section describes extensions to ordinary C grammar.

3.1 Operation Documentation

An operation definition can be preceded by an optional description in the form of a C string literal.

documented-definition ::= [C-string-literal] operation-definition

The use of a C string allows an implementation file to be processed by a C preprocessor without loosing the description, as happens with descriptions in the form of C comments. The preprocessor concatenates adjacent string literals, allowing a multi-line description to be written using multiple strings. Alternatively, a multi-line description can be written using '\' for line continuation. This description is stored in the operation data base where it can be extracted by documentation generation programs, that generators produce formatted documentation for Icon programmers and for C programmers maintaining the Icon implementation. The documentation generators are responsible for inserting newline characters at reasonable points when printing the description.

3.2 Types of Operations

rtt can be used to define the built-in functions, operators, and keywords of the Icon language. (Note that there are some Icon constructs that fall outside this implementation specification system. These include control structures such as string scanning and limitation, along with record constructors and field references.)

operation-definition ::=
function *result-seq* *identifier* (*[param-list]*) *[declare]* *actions* end |
operator *result-seq op* *identifier* (*[param-list]*) *[declare]* *actions* end |
keyword *result-seq* *identifier* *actions* end |
keyword *result-seq* *identifier* constant *key-const* end

result-seq ::=
{ *length* , *length [+]* } |
{ *length [+]* } |
{ }

length ::= integer | ∗

result-seq indicates the minimum and maximum length of the result sequence of an operation (the operation is treated as if it is used in a context where it produces all of its results). For example, addition always produces one result so its *result-seq* is {1, 1}. If the minimum and maximum are the same, only one number need be given, so the *result-seq* for addition can be coded as {1}. A conditional operation can produce either no results (that is, it can fail) or it can produce one result, so its *result-seq* is {0, 1}. A length of ∗ indicates an unbounded number of results, so the *result-seq* of ! is indicated by {0, ∗}. An ∗ in the lower bound means the same thing as 0, so {0, ∗} can be written as {∗, ∗}, which simplifies to {∗}. A *result-seq* of {} indicates no result sequence. This is not the same as a zero-length result sequence, {0}; an operation with no result sequence does not even fail. exit() is an example of such an operation.

A + following the length(s) in a *result-seq* indicates that the operation can be resumed to perform some side effect after producing its last result. All existing examples of such operations produce at most one result, performing a side effect in the process. The side effect on resumption is simply an undoing of the original side effect. An example of this is tab(), which changes &pos as the side effect.

For functions and keywords, *identifier* is the name by which the operation is known within the Icon language (for keywords, *identifier* does not include the &). For operations, *op* is (usually) the symbol by which the operation is known within the Icon language and *identifier* is a descriptive name. It is possible to have more than one operation with the same *op* as long as they have different identifiers and take a different number of operands. In all cases, the *identifier* is used to construct the name(s) of the C function(s) which implement the operation.

A *param-list* is a comma-separated list of parameter declarations. Some operations, such as write(), take a variable number of arguments. This is indicated by appending a pair of brackets enclosing an identifier to the last parameter declaration. This last parameter is then an array containing the *tail* of the argument list, that is, those arguments not taken up by the preceding parameters. The identifier in brackets represents the length of the tail and has a type of C integer.

param-list ::= param { , param }∗ [[identifier]]

Most operations need their arguments dereferenced. However, some operations, such as assignment, need undereferenced arguments and a few need both dereferenced and undereferenced versions of an argument. There are forms of parameter declarations to match each of these needs.

param ::= *identifier* |
underef *identifier* |
underef *identifier* –> *identifier*

A simple identifier indicates a dereferenced parameter. underef indicates an undereferenced parameter. In the third form of parameter declaration, the first identifier represents the undeferenced form of the argument and the second identifier represents the dereferenced form. This third form of declaration may not be used with the variable part of an argument list. These identifiers are of type *descriptor*. Descriptors are implemented as C structs. See [5] for a detailed explanation of descriptors.

Examples of operation headers:

```
"detab(s,i,...) – replace tabs with spaces, with stops at columns indicated."
function{1} detab(s, i[n])
   actions
end

"x <–> y – swap values of x and y."
" Reverses swap if resumed."
operator{0,1+} <–> rswap(underef x –> dx, underef y –> dy)
   declare
   actions
end
```

```
"&fail – just fail"
keyword{0} fail
    actions
end
```

3.3 Declare Clause

Some operations need C declarations that are common to several actions. These can be declared within the declare clause.

declare ::= declare { *C declarations* }

These may include *tended* declarations, which are explained below in the section on extensions to C. If a declaration can be made local to a block of embedded C code, it is usually better to put it there than in a declare clause. This is explained below in the discussion of the body action.

3.4 Constant Keywords

Any keyword can be implemented using general *actions*. However, for constant keywords, iconc sometimes can produce more efficient code if it treats the keyword as a literal constant. Therefore, a special declaration is available for declaring keywords that can be represented as Icon literals. The constant is introduced with the word constant and can be one of four literal types.

key-const ::= string-literal | cset-literal | integer-literal | real-literal

When using this mechanism, the programmer should be aware of the fact that rtt tokenizes these literals as C literals, even though they are later interpreted as Icon literals. This has only a few effects on their use. See [3] for a description of how the preprocessor interprets string literals during string concatenation; in some situations, escape sequences are explicitly treated as C characters. C does not recognize control escapes, so '\^', which is a valid Icon cset literal, is not recognized by rtt's C tokenizer, because the second quote ends the literal, leaving the third quote dangling. Only decimal integer literals are allowed.

3.5 Actions

All operations other than constant keywords are implemented with general *actions*.

Actions fall into four categories: type checking and conversions, detail code expressed in extended C, abstract type computations, and error reporting.

actions ::= { action }+

action ::= checking-conversions |
detail-code |
abstract { *type-computations* } |
runerr(*msg_number [* , *descriptor]*) *[;]*
{ *[actions]* }

3.5.1 Type Checking and Conversions

The type checking and conversions are

checking-conversions ::= if *type-check* then *action* |
if *type-check* then *action* else *action* |
type_case *descriptor* of { *{ type-select }+* }
arith_case (*descriptor* , *descriptor*) of { *arith-selections* }
len_case *identifier* of { *{ integer* : *action }+* default : *action* }

type-select ::= *{ type-name* : *}+ action* |
default : *action*

arith-selections ::= C_integer: *action* integer: *action* C_double: *action*

These actions specify run-time computations. These computations could be performed in C, but specifying them in the interface language gives the compiler information it can use to generate better code.

The if actions use the result of a *type-check* expression to select an action. The type_case action selects an action based on the type of a descriptor. If a type_case action contains a default clause, it must be last. *type-select* clauses must be mutually exclusive in their selection. The arith_case performs type conversion and type checking needed for a binary arithmetic operator. The statement

```
arith_case (x,y) of {
    C_integer:
        action 1
    integer:
        action 2
    C_double:
        action 3
    }
```

is functionally equivalent to

```
if cnv:(exact)C_integer(x) && cnv:(exact)C_integer(y) then
    action 1
#ifdef LargeInts
else if cnv:(exact)integer(x) && cnv:(exact)integer(y) then
    action 2
#endif                                          /* LargeInts */
else {
    if !cnv:C_double(x) then
        runerr(102, x)
    if !cnv:C_double(y) then
        runerr(102, y)
    abstract {
        return real
        }
    action 3
    }
```

but can sometimes be more implemented more efficiently. Note that the integer selection can only be taken if large integers are implemented. Otherwise, the C_integer selection is always taken for integer arithmetic.

The len_case action selects an action based on the length of the variable part of the argument list of the operation. The *identifier* in this action must be the one representing that length.

A *type-check* can succeed or fail. It is either an assertion of the type of a descriptor, a conversion of the type of a descriptor, or a logical expression involving *type-check*s. Only limited forms of logical expressions are supported.

type-check ::= *simple-check { && simple-check }** |
! *simple-check*

simple-check ::= is: *type-name* (*descriptor*) |
cnv: *dest-type* (*source [* , *destination]*) |
def: *dest-type* (*source* , *value [* , *destination]*)

dest-type ::= cset |
integer |
real |
string |
C_integer |
C_double |
C_string |
(exact)integer |
(exact)C_integer
tmp_string |
tmp_cset

The is check succeeds if the value of the descriptor is in the type indicated by *type-name*. Conversions indicated by cnv are the conversions between the Icon types of cset, integer, real, and string. Conversions indicated by def are the same conversions with a default value to be used if the original value is null.

dest-type is the type to which to a value is to be converted, if possible. cset, integer, real, and string constitute a subset of *icon-type* which is in turn a subset of *type-name* (see below). C_integer, C_string, and C_double are conversions to internal C types that are easier to manipulate than descriptors. Each of these types corresponds to an Icon type. A conversion to an internal C type succeeds for the same values that a conversion to the corresponding Icon type succeeds, except that a large integer cannot be converted to a C integer. C_integer represents the C integer type used for integer values in the particular Icon implementation being compiled (typically, a 32-bit integer type). C-double represents the C double type. C-string represents a pointer to a null-terminated C character array. However, see below for a discussion of the destination for conversion to C_string. (exact) before integer or C_integer disallows conversions from reals or strings representing reals, that is, the conversion fails if the value being converted represents a real value.

Conversion to tmp_string is the same as conversion to string (the result is a descriptor), except that the string is only guaranteed to exist for the lifetime of the operation (the lifetime of a suspended operation extends until it can no longer be resumed). Conversion to tmp_string is generally less expensive than conversion to string and is never more expensive, but the resulting string must not be exported from the operation. tmp_cset is analogous to tmp_string.

The source of the conversion is the descriptor whose value is to be converted. The type of the destination of the conversion depends on the type of conversion. The destinations for conversions to cset, integer, real, string, (exact)integer, tmp_string, and tmp_cset must be descriptors. The destinations for conversions to C_integer, C_double, and (exact)C_integer must be the corresponding C types. However, the destination for conversion to C_string must be tended. If the destination is declared as tended char *, then the d-word (string length) of the tended location will be set, but the operation will not have direct access to it. The variable will look like a char *. Because the operation does not have access to the string length, it is not a good idea to change the pointer once it has been set by the conversion. If the destination is declared as a descriptor, the operation has access to both the pointer to the string and the string's length (which includes the terminating null character).

If no destination is specified, the conversion is done in-place. However, conversions to C values require a destination with a different type from the source. For convenience, ''in-place'' conversions to C values are allowed for simple parameters, though not for other variables. For parameters, rtt creates a destination of the correct type and establishes a new meaning for the parameter name within the *lexical scope* of the conversion. Within this scope, the parameter name refers to the destination; in effect, the type of the parameter has been converted along with its value. In the case of conversion to C_string, the destination is a tended pointer.

The scope of an in-place conversion to a C value extends from the conversion along all execution paths that can be taken if the conversion succeeds. Along execution paths that can be taken if the conversion fails, parameters names retain their previous meanings. It is possible for two scopes for one parameter name to overlap. This is only a problem if the parameter is referenced within the conflicting scope. rtt issues an error message if this occurs.

The second argument to the def conversion is the default value. The default value can be any C expression that evaluates to the correct type. These types are given in the following chart.

```
cset:              struct b_cset
integer:           C_integer
real:              double
string:            struct descrip
C_integer:         C_integer
C_double:          double
C_string:          char *
tmp_string:        struct descrip
tmp_cset:          struct b_cset
(exact)integer:    C_integer
(exact)C_integer:  C_integer
```

The numeric operators provide good examples of how conversions are used. (Note that the following version of negation does not support large integers, because they cannot be converted to C integers.)

```
operator{1} – neg(x)
   if cnv:(exact)C_integer(x) then {
      actions
      }
   else {
      if !cnv:C_double(x) then
         runerr(102, x)
      actions
      }
end
```

Within the code indicated by *actions*, x refers to C values rather than to the Icon descriptors of the unconverted parameter.

In the case of conversions combined with &&, the scopes are also tied together. This occurs in the code given earlier as equivalent to the arith_case statement. Even though an else clause in that code may be taken after the conversion of x succeeds, the clause is in the scope of the unconverted x. For a true in-place conversion (where the result is a descriptor rather than a C value) there is no question of scope but there is a question of whether the value of x has been converted or not in the else clause. This depends on whether the conversion of x or the conversion of y caused the failure of the condition and may also depend on how rtt and iconc implement these expressions. rtt issues warning messages when there might be confusion or ambiguity.

The subject of any type check or type conversion must be an unmodified parameter. For example, once an in-place conversion has been applied to a parameter, another conversion may not be applied to the same parameter. This simplifies the clerical work needed by type inference in iconc. This restriction does not apply to type checking and conversions in C code.

3.5.2 Type Names

The *type-name*s represent types of Icon intermediate values, including variable references. These are the values that enter and leave an operation; ‘‘types’’ internal to data structures, such as list element blocks, are handled completely within the C code.

```
type-name ::=   empty_type |
                any_value |
                icon-type |
                variable-ref
```

icon-type ::= null |
string |
cset |
integer |
real |
file |
list |
set |
table |
record |
proc |
coexpr

variable-ref ::= variable |
tvsubs |
tvtbl |
kywdint |
kywdpos |
kywdsubj

The *type-name*s are not limited to the first-class types of Icon's language definition. The *type-name*s that do not follow directly from Icon types need further explanation. empty_type is the type containing no values and is needed for conveying certain information to the type inference system, such as an unreachable state. For example, the result type of stop is empty_type. It may also be used as the internal type of an empty structure. Contrast this with null, which consists of the null value. any_value is the type containing all first-class types, that is, those in icon-type.

Variable references are not first-class values in Icon; they cannot be assigned to variables. However, they do appear in the definition of Icon as arguments to assignments and as the subject of dereferencing. For example, the semantics of the expression

```
s[3] := s
```

can be described in terms of a substring trapped variable and a simple variable reference. For this reason, it is necessary to include these references in the type system of RTL. variable consists of all variable references. It contains five distinguished subtypes. tvsubs contains all substring trapped variables. tvtbl contains all table-element trapped variables. kywdint contains &error, &random, and &trace. kywdpos contains &pos. kywdsubj contains &subject.

3.5.3 Including C Code

As noted above, C declarations can be included in a declare clause. Embedded C code may reference these declarations as well as declarations global to the operation.

Executable C code can be included using one of two actions.

detail-code ::= body { *extended-C* } |
inline { *extended-C* }

For the interpreter, body and inline are treated the same. For the compiler, inline indicates code that is reasonable for the compiler to put in-line when it can; that is, this code is part of the in-line version of the operation that is put in the data base. body indicates that, for the in-line version of the operation, this piece of C code should be put in a separate function in the link library and the body action should be replaced by a call to that function. Any parameters of the operation or variables from the declare clause needed by the function must be passed as arguments to the function. Therefore, it is more efficient to declare variables needed by a body action within that body than within the declare clause. However, the scope of these local variables is limited to the body code.

Most Icon keywords provide examples of operations that should be generated in-line. In the following example, nulldesc is a global variable of type descriptor. It is defined in the header files automatically included by rtt.

```
"&null – the null value."
keyword{1} null
    abstract {
        return null
        }
    inline {
        return nulldesc;
        }
end
```

3.5.4 Error Reporting

runerr(*msg_number [, descriptor]*) *[;]*

runerr is translated into a call to the run-time error handling routine. Specifying this as a separate action rather than a C expression within a body or inline action gives the compiler additional information about the behavior of the operation. *msg_number* is the number used to look up the error message in a run-time error table (see the file runtime/data.r). If a descriptor is given, it is taken to be the offending value.

3.5.5 Abstract Type Computations

abstract { *type-computations* }

The behavior of an operation with respect to types is a simplification of the full semantics of the operation. For example, the semantics of the function image is to produce the string representing its operand; its behavior in the type realm is described as simply returning some string. In general, a good simplification of an operation is too complicated to be automatically produced from the operation's implementation (of course, it is always possible to conclude that an operation can produce any type and can have any side effect, but that is hardly useful). For this reason, the programmer must use the abstract action to specify *type-computation*s (for operations that appear only in an interpreter version of Icon, abstract type computations are optional).

type-computations ::= *{* store [*type*] = *type [;] }** *[* return *type [;]]*

type-computations consist of side effects and a statement of the result type of the operation. There must be exactly one return *type* along any path from the start of the operation to C code containing a return, suspend, or fail.

A side effect is represented as an assignment to the *store*. The store is analogous to program memory. Program memory is made up of locations containing values. The store is made up of locations containing types. A type represents a set of values, though only certain such sets correspond to types for the purpose of abstract type computations. Types may be basic types such as all Icon integers, or they may be composite types such as all Icon integers combined with all Icon strings. The rules for specifying types are given below. A location in the store may correspond to one location in program memory, or it may correspond to several or even an unbounded number of locations in program memory. The contents of a location in the store can be thought of as a conservative (that is, possibly overestimated) summary of values that might appear in the corresponding location(s) in program memory at run time.

Program memory can be accessed through a pointer. Similarly, the store can be indexed by a pointer type, using an expression of the form store[*type*], to get at a given location. An Icon global variable has a location in program memory, and a reference to such a variable in an Icon program is treated as a pointer to that location. Similarly, an Icon global variable has a location in the store and, during type inference, a reference to the variable is interpreted as a pointer type indexing that location in the store. Because types can be composite, indexing into the store with a pointer type may actually index several locations. Consider the following side effect

store[*type1*] = *type2*

Suppose during type inference *type1* evaluates to a composite pointer type consisting of the pointer types for several global variables. Then all corresponding locations in the store will be updated. If the above side effect is coded in the assignment operator, this situation might result from an Icon expression such as

```
every (x | y) := &null
```

In this example, it is obvious that both variables are changed to the null type. However, type inference can only deduce that at least one variable in the set is changed. Thus, it must assume that each could either be changed or left as is. It is only when the left hand side of the side effect represents a unique program variable that type inference knows that the variable cannot be left as is. In the current implementation of type inference, assignment to a single named variable is the only side effect where type inference recognizes that the side effect will definitely occur.

Indexing into the store with a non-pointer type corresponds to assigning to a non-variable. Such an assignment results in error termination. Type inference ignores any non-pointer types in the index type; they represent execution paths that don't continue and thus contribute nothing to the types of expressions.

A type in an abstract type computation is of the form

type ::= *type-name* |
type (*variable*) |
component-ref |
new *type-name* (*type { , type }**) |
store [*type*] |
type ++ *type* |
type ** *type* |
(*type*)

The type(*variable*) expression allows type computations to be expressed in terms of the type of an argument to an operation. This must be an unmodified argument. That is, the abstract type computation involving this expression must not be within the scope of a conversion. This restriction simplifies the computations needed to perform type inference.

This expression is useful in several contexts, including operations that deal with aggregate types. The type system for a program may have several sub-types for an aggregate type. The aggregate types are list, table, set, record, substring trapped variable, and table-element trapped variable. Each of these Icon types is a composite type within the type computations, rather than a basic type. Thus the type inferencing system may be able to determine a more accurate type for an argument than can be expressed with a *type-name*. For example, it is more accurate to use

```
if is:list(x) then
   abstract {
      return type(x)
      }
   actions
else
   runerr(108, x)
```

than it is to use

```
if is:list(x) then
   abstract {
      return list
      }
   actions
else
   runerr(108, x)
```

Aggregate values have internal structure. Aggregate types also need an internal structure that summarizes the structure of the values they contain. This structure is implemented with type components. These components are referenced using dot notation:

component-ref ::= type . component-name

component-name ::= lst_elem |
set_elem |
tbl_key |
tbl_val |
tbl_dflt |
all_fields |
str_var |
trpd_tbl

Just as values internal to aggregate values are stored in program memory, types internal to aggregate types are kept in the store. A component is a pointer type referencing a location in the store.

A list is made up of (unnamed) variables. The lst_elem component of a list type is a type representing all the variables contained in all the lists in the type. For example, part of the code for the bang operator is as follows, where dx is the dereferenced operand.

```
type_case dx of {
    list: {
        abstract {
            return type(dx).lst_elem
            }
        actions
        }
    ...
    }
```

This code fragment indicates that, if the argument to bang is in a list type, bang returns some variable from some list in that type. In the type realm, bang returns a basic pointer type.

The set_elem component of a set type is similar. The locations of a set never ‘‘escape’’ as variables. That is, it is not possible to assign to an element of a set. This is reflected in the fact that a set_elem is always used as the index to the store and is never assigned to another location or returned from an operation. The case in the code from bang for sets is

```
set: {
    abstract {
        return store[type(dx).set_elem]
        }
    actions
    }
```

Tables types have three components. tbl_key references a location in the store containing the type of any possible key in any table in the table type. tbl_val references a location containing the type of any possible value in any table in the table type. tbl_dflt references a location containing the type of any possible default value for any table in the table type. Only tbl_val corresponds to a variable in Icon. The others must appear as indexes into the store.

Record types are implemented with a location in the store for each field, but these locations cannot be accessed separately in the type computations of RTL. These are only needed separately during record creation and field reference, which are handled as special cases in the compiler. There is an RTL component name, all_fields, which is a composite type and includes the pointer types for each of the fields of a record type.

Substring trapped variables are implemented as aggregate values. For this reason, they need aggregate types to describe them. The part of the aggregate of interest in type inference is the reference to the underlying variable. This is reflected in the one component of these types, str_var. It is a reference to a location in the store containing the pointer types of the underlying the variables that are ‘‘trapped’’. str_var is only used as an index into the store; it is never exported from an operation.

Similarly table-element trapped variables need aggregate types to implement them. They have one component, trpd_tbl, referencing a location in the store containing the type of the underlying table. The key type is not kept separately in the trapped variable type; it must be immediately added to the table when a table-element trapped

variable type is created. This pessimistically assumes that the key type will eventually be put in the table, but saves a component in the trapped variable for the key. trpd_tbl is only used as an index into the store; it is never exported from an operation.

The type computation, new, indicates that an invocation of the operation being implemented creates a new instance of a value in the specified aggregate type. For example, the implementation of the list function is

```
function{1} list(size, initial)
   abstract {
      return new list(type(initial))
      }
   actions
end
```

The type arguments to the new computation specify the initial values for the components of the aggregate. The table type is the only one that contains multiple components. (Note that record constructors are created during translation and are not specified via RTL.) Table components must be given in the order: tbl_key, tbl_val, and tbl_dflt.

In the type system for a given program, a aggregate type is partitioned into several sub-types (these sub-types are only distinguished during type inference, not at run time). One of these sub-types is allocated for every easily recognized use of an operation that creates a new value for the aggregate type. Thus, the following Icon program has two list sub-types: one for each invocation of list.

```
procedure main()
   local x

   x := list(1, list(100))
end
```

Two operations are available for combining types. Union is denoted by the operator '++' and intersection is denoted by the operator '**'. Intersection has the higher precedence. These operations interpret types as sets of values. However, because types may be infinite, these sets are treated symbolically.

4. C Extensions

The C code included using the declare, body, and inline actions may contain several constructs beyond those of standard C. There are five categories of C extensions: access to interface variables, declarations, type conversions/type checks, signaling run-time errors, and return statements.

In addition to their use in the body of an operation, the conversions and type checks, and declaration extensions may be used in ordinary C functions that are put through rtt.

4.1 Interface Variables

Interface variables include parameters, the identifier for length of the variable part of an argument list, and the special variable result. Unconverted parameters, converted parameters with Icon types, and converted parameters with the internal types tmp_string and tmp_cset are descriptors and within the C code have the type struct descrip. Converted parameters with the internal type of C_integer have some signed integer type within the C code, but exactly which C integer type varies between systems. This type has been set up using a typedef in one of the header files automatically included by rtt, so it is available for use in declarations in C code. Converted parameters with the internal type of C_double have the type double within the C code. Converted parameters of the type C_string have the type char *. The length of the variable part of a argument list has the type int within the C code.

result is a special descriptor variable. Under some circumstances it is more efficient to construct a return value in this descriptor than to use other methods. See Section 5 for details.

4.2 Declarations

The extension to declarations consists of a new storage class specifier, tended (register is an example of an existing storage class specifier). Understanding its use requires some knowledge of Icon storage management. Only a brief description of storage management is given here; see [5] for details.

Icon values are represented by descriptors. A descriptor contains both type information and value information. For large values (everything other than integers and the null value) the descriptor only contains a pointer to the value, which resides elsewhere. When such a value is dynamically created, memory for it is allocated from one of several memory regions. Strings are allocated from the *string region*. All other relocatable values are allocated from the *block region*. The only non-relocatable values are co-expression stacks and co-expression activation blocks. On some systems non-relocatable values are allocated in the *static region*. On other systems there is no static region and these values are allocated using the C malloc() function.

When a storage request is made to a region and there is not enough room in that region, a *garbage collection* occurs. All *reachable* values for each region are located. Values in the string and block regions are moved into a contiguous area at the bottom of the region, creating (hopefully) free space at the end of the region. Unreachable co-expression stacks and activator blocks are ‘‘freed’’. The garbage collector must be able to recognize and save all values that might be referenced after the garbage collection and it must be able to find and update all pointers to the relocated values. Operation arguments that contain pointers into one of these regions can always be found by garbage collection. The implementations of many operations need other descriptors or pointers into memory regions. The tended storage class identifies those descriptors and pointers that may have *live* values when a garbage collection could occur (that is, when a memory allocation is performed).

A descriptor is implemented as a C struct named descrip, so an example of a tended descriptor declaration is

```
tended struct descrip d;
```

Blocks are also implemented as C structs. The following list illustrates the types of block pointers that may be tended.

```
tended struct b_real *bp;
tended struct b_cset *bp;
tended struct b_file *bp;
tended struct b_proc *bp;
tended struct b_list *bp;
tended struct b_lelem *bp;
tended struct b_table *bp;
tended struct b_telem *bp;
tended struct b_set *bp;
tended struct b_selem *bp;
tended struct b_slots *bp;
tended struct b_record *bp;
tended struct b_tvkywd *bp;
tended struct b_tvsubs *bp;
tended struct b_tvtbl *bp;
tended struct b_refresh *bp;
tended struct b_coexpr *cp;
```

Alternatively, a union pointer can be used to tend a pointer to any kind of block.

```
tended union block *bp;
```

Character pointers may also be tended. However, garbage collection needs a length associated with a pointer into the string region. Unlike values in the block region, the strings themselves do not have a length stored with them. Garbage collection treats a tended character pointer as a zero-length string (see cnv:C_string for an exception). These character pointers are almost always pointers into some string, so garbage collection effectively treats them as zero-length substrings of the strings. The string as a whole must be tended by some descriptor so that it is preserved. The purpose of tending a character pointer is to insure that the pointer is relocated with the string it points into. An example is

```
tended char *s1, *s2;
```

Tended arrays are not supported. tended may only be used with variables of local scope. tended and register are mutually exclusive. If no initial value is given, one is supplied that is consistent with garbage collection.

4.3 Type Conversions/Type Checks

Some conditional expressions have been added to C. These are based on type checks in the type specification part of interface portion of RTL.

```
is: type-name ( source )
cnv: dest-type ( source , destination )
def: dest-type ( source , value , destination )
```

source must be an Icon value, that is, a descriptor. *destination* must be a variable whose type is consistent with the conversion. These type checks may appear anywhere a conditional expression is valid in a C program. Note that is, cvn, and def are reserved words to distinguish them from labels.

The type_case statement may be used in extended C. This statement has the same form as the corresponding action, but in this context, C code replaces the *action*s in the *type-select* clauses.

4.4 Signaling Run-time Errors

runerr() is used for signaling run-time errors. It acts like a function but may take either 1 or 2 arguments. The first argument is the error number. If the error has an associated value, the second argument is a descriptor containing that value. runerr() automatically implements error conversion when it is enabled.

4.5 Return Statements

There are four statements for leaving the execution of an operation:

```
ret-statments ::= return ret-value ; |
                  suspend ret-value ; |
                  fail ; |
                  errorfail ;
```

The first three are analogous to the corresponding expressions in the Icon language. errorfail acts like fail, but the compiler assumes it is unreachable when error conversion is disabled. It is used following error functions, such as irunerr(), that do not automatically implement error conversion. Note that it is not needed after the special construct runerr().

Return values can be specified in several ways:

```
ret-value ::=   descriptor |
                C_integer expression |
                C_double expression |
                C_string expression |
                descript-constructor
```

descriptor is an expression of type struct descrip. For example

```
{
    tended struct descrip dp;
    ...
    suspend dp;
    ...
}
```

Use of C_integer, C_double, or C_string to prefix an expression indicates that the expression evaluates to the indicated C type and not to a descriptor. When necessary, a descriptor is constructed from the result of the expression, but when possible the Icon compiler produces code that can use the raw C value (See Section 5). As an example, the integer case in the divide operation is simply:

```
inline {
   return C_integer x / y;
   }
```

Note that a returned C string must not be in a local (dynamic) character array; it must have a global lifetime.

A *descript-constructor* is an expression that explicitly converts a pointer into a descriptor. It looks like a function call, but is only valid in a return statement.

descript-constructor ::= string (*length* , *char-ptr*) |
cset (*block-ptr*) |
real (*block-ptr*) |
file (*block-ptr*) |
procedure (*block-ptr*) |
list (*block-ptr*) |
set (*block-ptr*) |
record (*block-ptr*) |
table (*block-ptr*) |
co_expression (*stack-ptr*) |
tvtbl (*block-ptr*) |
named_var (*descr-ptr*) |
struct_var (*descr-ptr* , *block-ptr*) |
tvsubs (*descr-ptr* , *start* , *len*) |
kywdint (*descr-ptr*) |
kywdpos (*descr-ptr*) |
kywdsubj (*descr-ptr*)

The arguments to string are the length of the string and the pointer to the start of the string. *block-ptr*s are pointers to blocks of the corresponding types. *stack-ptr* is a pointer to a co-expression stack. *descr-ptr* is a pointer to a descriptor. named_var is used to create a reference to a variable (descriptor) that is not in a block. struct_var is used to create a reference to a variable that is in a block. The Icon garbage collector works in terms of whole blocks. It cannot preserve just a single variable in the block, so the descriptor referencing a variable must contain enough information for the garbage collector to find the start of the block. That is what the *block-ptr* is for. tvsubs creates a substring trapped variable for the given descriptor, starting point within the string, and length. kywdint, kywdpos, and kywdsubj create references to keyword variables.

Note that returning either C_double *expression* or tvsubs(*descr-ptr*, *start*, *len*) may trigger a garbage collection.

5. Programming Considerations

In general, an operation can be coded in many ways using RTL. Some codings are obviously better than others, but details of how the rtt and iconc interact to produce efficient generated code can produce subtle effects.

5.1 The Use of ‘‘Result’’

A person coding an operation for the run-time system can always treat result as if it exists. However, the Icon compiler tries to minimize the number of locations needed for intermediate results. If the programmer does not reference result, the compiler knows that the result location of the operation is only written to after all references to parameters are completed (in the absence of backtracking and resumption), so it may overlap the result location with a parameter location. Thus not referencing result may result in more space efficient code. On the other hand, it is a bad idea to explicitly declare a tended descriptor, compute the result value into it, and then return the declared descriptor. The reason is that returning an ordinary descriptor requires that the value be copied from one descriptor to another. If result is returned, the translator knows that the value is already where it needs to be and generates no instructions to do a copy. In addition, there is overhead for having explicitly declared tended descriptors within a library routine (although the extra overhead is minimal for in-line code). The rule is: if a named location really is needed in which to compute the result, use result. An example of this occurs in the code for copy when a utility routine written in standard C is used to make a copy of a list.

```
type_case x of {

    list:
        inline {
            if (cplist(&x, &result, (word)1, BlkLoc(x)–>list.size + 1) == Error)
                runerr(0);
            return result;
            }
    ...
```

Note that BlkLoc is a macro which accesses the block pointer in a descriptor. It is defined in Icon's standard header files.

5.2 Returning C Values Vs. Icon Values

For operations that produce integers or reals, the programmer has a choice of coding the return statement using a C value or an Icon value. For a returned string, the choice only exists when the string can be represented as a null terminated string.

In general, a returned C value will have to be converted to a descriptor. However, when the value is immediately used as an argument to an operation which converts it back to the C value, it is a waste of time and resources to convert it the descriptor. This frequently happens in arithmetic. If the conversion of the result is explicit, for example in:

```
return C_double n + m;
```

iconc can recognize the conversion and cancel it with the conversion in the operation using the value. This makes the explicit conversion the better choice. (This optimization is not currently implemented in iconc, but operations should be coded to take advantage of it when it is implemented.)

5.3 Warnings

rtt does not do type checking on general C expressions. It passes these on to the C compiler. By default, rtt produces #line directives in its output relating that code back to the source from which it came, but this process is not perfect. Thus errors in type compatibility detected by the C compiler may appear somewhat removed from the position in the original source.

In the presence of tended variables, rtt must produce untending code at function return sites. If a return occurs in a macro, that macro must be expanded by rtt's preprocessor. It must not be in a header file that is only expanded when the C compiler is run. Failure to follow this rule may result in run-time failures that are difficult to diagnose.

When generating code for the compiler run-time system, rtt splits operation code into separate files C files. These operations do not have access to static declarations in the original source file. Such static declarations must be avoided.

rtt supports ANSI standard C. On some systems, system header files make use of extensions to C. For portability, system header files should be included by the C compiler, not rtt.

Acknowledgements

The design of RTL came out of discussions with Ralph Griswold concerning the features needed in an Icon implementation language for use with an optimizing compiler. Clint Jeffery was the first RTL programmer and provided feedback on the language, its implementation, and its documentation. Gregg Townsend also participated in discussions of RTL and its features.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.

2. *American National Standard for Information Systems — Programming Language - C, ANSI X3.159-1989*, American National Standards Institute, New York, 1990.

3. K. Walker, *A Stand-Alone C Preprocessor*, The Univ. of Arizona Icon Project Document IPD65a, 1989.

4. R. E. Griswold, *Supplementary Information for the Implementation of Version 8.5 of Icon*, The Univ. of Arizona Icon Project Document IPD180, 1992.

5. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.

6. R. E. Griswold, C. L. Jeffery, G. M. Townsend and K. Walker, *Installing the Icon Version 8.7 Interpreter on UNIX Systems*, The Univ. of Arizona Icon Project Document IPD175, 1992.

Appendix A: RTL Grammar

documented-definition ::= *[C-string-literal] operation-definition*

operation-definition ::=
function *result-seq* *identifier* (*[param-list]*) *[declare]* *actions* end |
operator *result-seq op* *identifier* (*[param-list]*) *[declare]* *actions* end |
keyword *result-seq* *identifier* *actions* end |
keyword *result-seq* *identifier* constant *key-const* end

result-seq ::=
{ *length* , *length [+]* } |
{ *length [+]* } |
{ }

length ::= *integer* | *

param-list ::= *param { , param }* [[*identifier*]]

param ::=
identifier |
underef *identifier* |
underef *identifier* –> *identifier*

declare ::= declare { *C declarations* }

key-const ::= *string-literal* | *cset-literal* | *integer-literal* | *real-literal*

actions ::= *{ action }+*

action ::=
checking-conversions |
detail-code |
abstract { *type-computations* } |
runerr(*msg_number [, descriptor]*) *[;]*
{ *[actions]* }

checking-conversions ::=
if *type-check* then *action* |
if *type-check* then *action* else *action* |
type_case *descriptor* of { *{ type-select }+* }
arith_case (*descriptor* , *descriptor*) of { *arith-selections* }
len_case *identifier* of { *{ integer* : *action }+* default : *action* }

type-select ::=
{ type-name : *}+ action* |
default : *action*

arith-selections ::= C_integer: *action* integer: *action* C_double: *action*

type-check ::=
simple-check { && simple-check } |
! *simple-check*

simple-check ::=
is: *type-name* (*descriptor*) |
cnv: *dest-type* (*source [, destination]*) |
def: *dest-type* (*source* , *value [, destination]*)

dest-type ::=
cset |
integer |
real |
string |
C_integer |
C_double |
C_string |
(exact)integer |
(exact)C_integer
tmp_string |
tmp_cset

type-name ::=
empty_type |
icon-type |
variable-ref

icon-type ::=
null |
string |
cset |
integer |
real |
file |
list |
set |
table |
record |
procedure |
co_expression

variable-ref ::=
variable |
tvsubs |
tvtbl |
kywdint |
kywdpos |
kywdsubj

detail-code ::=
body { *extended-C* } |
inline { *extended-C* }

type-computations ::=
{ store [*type*] = *type [;] }** *[* return *type [;]]*

type ::=
type-name |
type (*variable*) |
component-ref |
new *type-name* (*type { , type }**) |
store[*type*] |
type ++ *type* |
type ** *type* |
(*type*)

component-ref ::=
type . component-name

component-name ::= lst_elem |
set_elem |
tbl_key |
tbl_val |
tbl_dflt |
all_fields |
str_var |
trpd_tbl

Appendix B: Extensions to C

Declarations:

```
tended struct descrip    d;
tended struct b_real     *bp;
tended struct b_cset     *bp;
tended struct b_file     *bp;
tended struct b_proc     *bp;
tended struct b_list     *bp;
tended struct b_lelem    *bp;
tended struct b_table    *bp;
tended struct b_telem    *bp;
tended struct b_set      *bp;
tended struct b_selem    *bp;
tended struct b_slots    *bp;
tended struct b_record   *bp;
tended struct b_tvkywd   *bp;
tended struct b_tvsubs   *bp;
tended struct b_tvtbl    *bp;
tended struct b_refresh  *bp;
tended struct b_coexpr   *cp;
tended union block       *bp;
tended char *s;
```

Conditional expressions:

```
is: type-name ( source )
cnv: dest-type ( source , destination )
def: dest-type ( source , value , destination )
```

Statements:

```
type_case descriptor of {
        { { type-name : }+ C-statment }+
        [ default: C-statment ]
        }
runerr( errnum );
runerr( errnum , descriptor  );
return ret-value ;
suspend ret-value ;
fail ;
errorfail ;
```

Where *ret-value* is one of:

```
descriptor
C_integer expression
C_double expression
C_string expression
string ( length , char-ptr )
cset ( block-ptr )
real ( block-ptr )
file ( block-ptr )
procedure ( block-ptr )
list ( block-ptr )
```

set (*block-ptr*)
record (*block-ptr*)
table (*block-ptr*)
co_expression (*stack-ptr*)
tvtbl (*block-ptr*)
named_var (*descr-ptr*)
struct_var (*descr-ptr* , *block-ptr*)
tvsub (*descr-ptr* , *start* , *len*)
kywdint (*descr-ptr*)
kywdpos (*descr-ptr*)
kywdsubj (*descr-ptr*)

Appendix C: Reserved Words beyond those of ANSI Standard C

C_integer is not ‘‘fully’’ reserved because it needs to be defined with a typedef to the C integer type used to implement Icon integers. This ability is needed to be able to write system independent code. C_double and C_string are treated the same for consistency, but typedefs are not actually needed for them.

C_double	(may be defined with a typedef)
C_integer	(may be defined with a typedef)
C_string	(may be defined with a typedef)
cnv	
def	
deref	
errorfail	
fail	
function	
is	
keyword	
operator	
runerr	
suspend	
tended	
type_case	

In addition, identifiers starting with r_ are reserved for variables generated by the translator and should not be used in RTL code.

Appendix D: Using rtt

rtt is invoked with the command

rtt *{ option }* { file }+*

One or more file names must be specified. If a name lacks the .r suffix, it is appended. A file name of – indicates standard input; when needed, the name stdin is used for constructing an output file name.

The rtt options are

–C – Retain comments and all white space; only effective with –E

–D *identifier [= [text]]* – Predefine an identifier for the preprocessor.

–E – Run only the preprocessor, sending the result to standard output.

–I *path* – Use *path* as one of the standard locations when searching for header files. Because rtt does not usually processes system header files, this preprocessor option is not generally used; see the –r option.

–P – Suppress #line directives in the output. The default, both with and without –E, is to place #line directives in the output. These directives relate the code back to its source location in the input file.

–U *identifier* – Undefine an identifier that is predefined in the preprocessor.

–d *file* – Use *file* as the data base. A suffix of .db is added if it is missing. The standard data base is rt.db.

–r *path* – Use *path* to locate the C header files for the Icon system. rtt locates header files by appending ../src/h/*header-file* to *path*; *path* must end with /. The default path is established when rtt is compiled; see [6]. –r is useful for cross compiling the run-time system for another machine.

–t *name* – Treat *name* as a typedef name while parsing the .r files (alternately a dummy typedef can be put in h/grttin.h). This is useful when the actual typedef is in a system include file that is only included by the C compiler and not rtt.

–x – Produce code for iconx and don't update a data base; the default is to produce code for iconc.

Input files are translated into standard C. When translating a file for the interpreter, the C code is put in a file whose name is constructed by prepending x to the input file name and replacing the .r suffix by .c. No other files are created or updated for the interpreter.

When translating a file for the compiler, the C code is placed in several files (this allows more selective linking of run-time operations with compiled programs). Each operation is placed in a separate file whose name is generated by rtt. Any non-operation code is placed in a file whose name is constructed by replacing the .r suffix of the input file name by .c (note that this does not conflict with the output file created for the interpreter; it has an x prepended). For the compiler, rtt also places information about operations the data base. In addition, dependency information relating generated files names to the corresponding sources files is also stored in the data base (see Appendix E).

If the data base is non-existent, it is created. If an operation that already exists in the data base is retranslated, its entry in the data base is updated. rtt generates a unique *prefix* for every operation. The name of a C function implementing an operation is created by prepending a one-character *operation code* (this indicates whether the operation is a function, keyword or operator) and the unique prefix to the operation name. For example, the Icon function write() might be implemented by the C function F1d_write. Every body function for an operation is assigned a third prefix character to distinguish it. These generated names help prevent conflicts with the names of

existing C library functions. The prefixes are also used in creating the C file names; the C function for write() is put in the file f_1d.c. If an operation is retranslated, the original prefix is re-used.

When translating RTL for the compiler, rtt produces two files containing lists of file names. rttcur.lst is a list of output files created in the current execution. rttfull.lst is a list of all output files listed in the data base; that is, all files produced since the data base was created. rttfull.lst excludes files that contain no storage definitions; that is, those that only contain prototypes, typedefs, and externs. Both lists consists names without suffixes. See runtime/Makefile for an example of how these lists are used to create and maintain an Icon compiler run-time library.

Appendix E: Data Base Format

The data base is stored externally as a text file. This data base is designed to be easily processed by programs. The format of the data base is not designed for human interpretation, but a human being can read it with only minimal difficulty. The data base contains some redundant information where the information is more convenient to store than recompute.

The data base begins with some brief version information and contains six sections: types, components, functions, operators, keywords, and dependencies. Each section starts with a header consisting of the corresponding name and ends with $endsect. The type section relates data base type codes to type names. The codes consist of T followed by an integer. The entries in this section consist of the code and type name separated by a colon. The component section is similar; it relates type component codes to component names. These codes begin with C.

The next three sections contain implementation information for operations. The entries in these sections have the same format, except that each entry in the operators section has the operator symbol prepended. The implementation entries are stored in alphabetical order within each section.

Operation Entries

The following grammar describes the format of an implementation entry (without the prepended operator symbol for entries in the operator section). Most of this grammar closely reflects the grammar for RTL, except that much of the information is in prefix format. C-style comments are used in the grammar to explain non-terminals for otherwise obscure codes. Productions marked with † do not contain embedded white space.

implementation ::= header description tended-vars nontended-vars action $end

In addition to information obtained from the implementation of the operation, the *header* contains an assigned prefix that is unique within the type of operation (i.e. within functions, operators, or keywords).

header ::= name prefix params result-seq return-stms explicit-result

params includes the number of parameters to the operation followed by a list of codes indicating whether each parameter is dereferenced and/or undereferenced and indicating, by the presence or absence of a trailing v, whether the last parameter represents the tail of a variable length argument list.

params ::= †num-params (*[param-ind { , param-ind }∗ [v]]*)

num-params ::= integer

param-ind ::= u | /∗ undereferenced ∗/
d | /∗ dereferenced ∗/
du /∗ both dereferenced and undereferenced ∗/

result-seq is the same as the production in RTL, except that there is never just one bound (though there may be zero) and ∗ is never used in the lower bound.

result-seq ::= †{ *[integer , length [+]]* }

length ::= integer | ∗

The operation description is taken directly from the description in the RTL code. If no description is given, the empty string is used.

description ::= string-literal

return-stms is a sequence of four codes indicating whether the operation contains fail, return, suspend, or errorfail (possibly an implicit errorfail for a runerr) statements. An underscore in a position indicates that the corresponding statement is not used in the operation.

return-stms ::= †fail return suspend errorfail

fail ::= f | _

return ::= r | _

suspend ::= s | _

errorfail ::= e | _

explicit-result indicates whether the operation explicitly references the result location. It is either true, t, or false, f.

explicit-result ::= t | f

tended-vars is the number of tended variables from the declare clause followed by a list of types and initial values for those variables. A *ptr-type* of * indicates the union block type. *tend-init* of nil indicates no explicit initial value.

*tended-vars ::= num-tended { tended-var }**

num-tended ::= integer

tended-var ::= tend-type tend-init

tend-type ::= desc | str | blkptr *ptr-type*

ptr-type ::= * |
b_real |
b_cset |
b_file |
b_proc |
b_list |
b_lelem |
b_table |
b_telem |
b_set |
b_selem |
b_slots |
b_record |
b_tvkywd |
b_tvsubs |
b_tvtbl |
b_refresh |
b_coexpr

tend-init ::= C-code | nil

nontended-vars is the number of non-tended variables from the declare clause followed by a list of the variable names and their declarations.

*nontended-vars ::= num-vars { nontended-var }**

num-vars ::= integer

nontended-var ::= identifier C-declaration

C-declaration ::= C-code

action is very similar to that nonterminal in the RTL syntax. nil indicates no code. const indicates a keyword constant. if1 indicates an if-then action. if2 indicates and if-then-else action. case1 indicates a type_case action with no default clause. case2 indicates a type_case action with a default clause. acase indicates and arith_case action. lcase indicates a len_case action. abstr indicates a abstract action. block indicates in-line C code. It includes an indication of the tended locals needed by the code.

call indicates a call to a run-time library function implementing a body action. A third prefix character is assigned to each block to distinguish its function from that of other blocks and from the most general function for the operation. Other nonterminals in the production are described below.

err1 indicates a runerr with no value. err2 indicates a runerr with a value. lst indicates a sequence of actions.

action ::= nil |
const *type-code literal* |
if1 *type-check action* |
if2 *type-check action action* |
tcase1 *variable num_cases { typ-case }** |
tcase2 *variable num_cases { typ-case }* default* |
acase *variable variable action action action* |
lcase *num-cases { length-selection action }* default* |
abstr *[side-effects] [return-type]*
block *num-local-tended { local-tend-type }* C-code*
call *third-prefix return-value exit-codes use-result num-string-bufs num-cset-bufs num-args*
*{ arg-decl arg }**
err1 *integer* |
err2 *integer variable* |
lst *action action*

literal ::= string-literal | *cset-literal* | *integer-literal* | *real-literal*

num-cases ::= integer

typ-case ::= num-types { type-code }+ action

num-types ::= integer

length-selection ::= integer

default ::= action

type-check corresponds to that nonterminal in the RTL syntax. A suffix of 1 on the conversions indicates no explicit destination and a suffix of 2 indicates the presence of an explicit destination.

type-check ::= cnv1 *type-code source* |
cnv2 *type-code source destination* |
def1 *type-code source default-val* |
def2 *type-code source default-val destination* |
is *type-code variable* |
! *type-check* |
&& *type-check type-check*

source ::= variable

destination ::= C-code

default-val ::= C-code

The *symbol table* for an operation consists of the parameters, and both the tended and non-tended variables from the

declare clause of the operation. The symbol table is ordered by the order of appearance of the variables in the operation's entry in the data base. This ordering is used to determine the symbol table indexes. The variables are identified in the body of the operation entry by these indexes, with a special index of r indicating the special variable result. Subscripting is introduced by the prefix operator [.

variable ::= *symbol-table-index* |
[*symbol-table-index subscripting-index*

symbol-table-index ::= integer | r

subscripting-index ::= integer

A block of in-line C code is prepended with a list of the types of tended variables that must be allocated for the code. Any initializations are in the C code. These variables are referenced in the C code by positional index.

local-tend-type ::= desc | str | blkptr

A call to a body function contains information describing the interface to the function. In the standard calling convention, the function returns a signal and the result is return through a pointer to a descriptor. However, if only one signal is possible, a body function returns no signal. If, in addition, the result is either a C integer or a C double, the result is returned directly as the result of the function.

return-value ::= i | /* integer */
d | /* double */
n | /* no value */
s /* signal */

It is possible to *fall through* the end of a body function without ever encountering a return, suspend, or fail. This is reflected in the *exit-codes* for the body function.

exit-codes ::= †fail return suspend errorfail fall-through

fall-through ::= t | _

use-result indicates whether a pointer to a result descriptor must be passed to the function. It is either true, t, or false, f.

use-result ::= t | f

If the body function performs conversions to temporary strings or csets and the converted values must outlive the body function (they cannot outlive the operation, though) buffers must be allocated and passed to the body function. *num-string-bufs* and *num-cset-bufs* indicate the number of buffers needed.

num-string-bufs ::= integer

num-cset-bufs ::= integer

Other arguments to the body function are given explicitly. Declarations are given for the parameters for use in generating a prototype for the function.

arg-decl ::= C-code

arg ::= C-code

In RTL, side effects in the abstract clause are specified by expressions of the form store[*type*] = *type*. In the data base, they are represented using = as a prefix operator followed by the two types. The return type from an abstract clause is stored in the data base without the return.

side-effects ::= side-effect | lst *side-effects side-effect*

side-effect ::= = *variable-type value-type*

variable-type ::= type

value-type ::= type

return-type ::= type

The *type* nonterminal corresponds to the same one in the RTL syntax. typ indicates a *type-name* from RTL. The production vartyp *variable* comes from the type(*variable*) production of the RTL syntax. The new production differs in that the number of arguments is given explicitly rather than being indicated by parentheses. Other productions are simple prefix forms of the ones from RTL.

```
type ::=    typ type-code |
            vartyp variable |
            . type component-code |
            new type-code num-args { type }+ |
            store type |
            ++ type type |
            ** type type

component-code ::= †Cinteger |      /* see component section of data base */
                   f                /* all_fields */
```

type-code is used in many productions. It is clear from the corresponding RTL constructs which codes are valid in which contexts.

```
type-code ::= †Tinteger |  /* see type section of data base */
              e |          /* empty type */
              v |          /* variable */
              ci |         /* C integer */
              cd |         /* C double */
              cs |         /* C string */
              ei |         /* exact integer */
              eci |        /* exact C integer */
              ts |         /* temp string */
              tc |         /* temp cset */
              d |          /* return descriptor */
              nv |         /* return named variable */
              sv |         /* return structure variable */
              rn           /* return nothing explicitly */
```

C-code consists of text between the delimiters $c and $e. This text contains some special constructs all of which are introduced with $. Other text is assumed to be ordinary C code. Note that some special constructs contain sub-fields that are themselves C code. Therefore, $c-$e pairs may be properly nested.

$r indicates a non-modifying reference to a variable in the symbol table for the operation. $m indicates a modifying reference. $t indicates a reference to a tended variable local to the in-line block of code. $r and $m can be modified by an optional d. This is used with tended pointers to indicate that the entire descriptor must be referenced rather than just the pointer in the vword. $sb and $cb refer to string buffers and cset buffers respectively; each occurrence indicates a different buffer that needs to be allocated. $ret, $susp, and $fail represent return, suspend, and fail statements respectively. $efail represents the errorfail statement.

Several constructs are distinguished by special syntax so the peephole optimizer can locate them. $goto represents a C goto statement. $cgoto represents a conditional goto; the condition is presented as a piece of C code. $lbl introduces a label. ${ and $} are brackets. They are distinguished so the peephole optimizer does not eliminate a right bracket but leaves the left bracket when the end of a block is unreachable.

special-constructs ::= *†*$r *[* d *]* *symbol-table-index* |
†$m *[* d *]* *symbol-table-index* |
†$t *symbol-table-index* |
$sb |
$cb |
$ret *ret-value* |
$susp *ret-value* |
$fail |
$efail |
$goto *label* |
$cgoto *C-code label* |
$lbl *label* |
${ |
$}

ret-value corresponds to the same non-terminal in the RTL syntax. A uniform representation is used in the data base with the number of subexpressions explicitly given.

ret-value ::= type-code num-subexpr { C-code }+

Labels are represented as integers; C identifiers must be allocated for those that are not optimized away.

label ::= integer

Dependencies

The final section in the data base is the dependencies section. It has an entry for each RTL source file. Each entry starts with the source file name and is followed by a list of C files that depend on the RTL source file. The elements of the list are separated by white space. Each list ends with $end.

Appendix F: Adding Types to Icon

This Appendix describes how to add new types to Icon. It deals with simple types, aggregate types whose values contain other Icon values, and keywords that are variables. These are the kinds of types that are most likely to be added to the language. Executable types such as procedures and co-expressions are beyond the scope of this appendix as are types that require special representations and variable types that have special dereferencing semantics.

The Implementation of Icon Types

An Icon value is implemented as a two-word *descriptor* containing type information and value information [5]. The first word of a descriptor is the *dword*. For the types discussed here, the dword contains a unique code that identifies the type (see [5] for types that have additional information in the dword). The second word of the descriptor is the *vword*; it contains the actual value or a reference to the value.

Actual values that are too large to fit in one word are usually put in the block region. This region is controlled by a storage management system that includes a garbage collector. The garbage collector is driven by information in arrays indexed using the type codes associated with the blocks. The block region contains values for both simple and aggregate types.

There are several other parts of the run-time system besides the garbage collector that need information about Icon types. Some are Icon operations such as the type() function, while others are automaticly invoked features such as error trace back. These are described in more detail below. Types, of course, typically have operations associated with them that create and use values of the type.

The Type Specification System

Icon types are used in several places in RTL and new types must be added to this language. These uses include type checking constructs, return/suspend statements, and abstract type computations. In addition, the Icon compiler needs information about types in order to perform type inferencing. These requirements are satisfied with a type specification system.

This system is a simple declarative language for naming types and describing some of their properties. Information from the type specification system is incorporated in rtt and in iconc when they are built.

All types specified by the system may be used in the RTL is and type_case constructs. They may also be used in abstract type computations. Aggregate types may be used in a new type expression in an abstract type computation. A type specification may optionally indicate that RTL supports a special form of return/suspend statement that constructs a return value, in the form of a full descriptor, from a C value for the vword of the descriptor.

Type specifications are in the file common/typespec.txt. Comments in the file start with # and continue to the end of the line. This file is translated into a C header file by the program typespec. This is not part of the normal Icon build process; entries at the end of common/Makefile must be uncommented if typespec.txt is updated.

A type definition in the specification system has the form:

type-def ::= *identifier opt-abrv* : *kind opt-return*

where *identifier* is the name of the type and *opt-abrv* is an optional abbreviation for the type name. The abbreviation has the form

opt-abrv ::= *nil* |
{ *identifier* }

The abbreviation is used in tracing type inferencing and other places where a compact notation is desired. If no abbreviation is given, the full type name is used.

There are three *kind*s of types: simple, aggregate, and variable. Their syntax and usage are described in separate sections below. *opt-return* indicates optional RTL return/suspend support for the type. The four types of vwords supported by this construct are introduced below as needed. A complete grammar for the specification language is given near the end of this appendix.

Simple Value Types

Types with a *kind* clause of simple are simple in the sense that values of the type do not have components that contain other Icon values. These types may otherwise have sophisticated semantics.

There are three ways to implement the values of a type: encode them as C integers (these are guaranteed to be at least 32 bits long), implement them as blocks in the block region, or implement them in storage allocated using malloc() (in theory values can also be put in the string region, but it is only well suited for Icon strings; descriptors pointing into this region must have a special form). The choice of implementation determines the type of C value stored in the vword of the descriptor representing the Icon value.

The dword of a descriptor for one of these types contains a fixed code. It consists of a small integer type code along with flags describing characteristics of the descriptor. The small integer is represented by a preprocessor constant defined in h/rmacros.h. Its name is created by capitalizing the first character of the type name and prepending T_. For example, the definition of the type code for the cset type is

```
#define T_Cset 4
```

These definitions must be manually added to h/rmacros.h and the constant MaxType must be updated if the integer exceeds the current value of MaxType.

There are corresponding constants with names beginning with D_ that include both the T_ type code and the flags required for the type. For the types dealt with here, the flags in the constant D_Typecode must always be included, and if the vword points to storage under control of the garbage collector, the flag F_Ptr must be included. The definition of the D_ constant for the cset type is

```
#define D_Cset (T_Cset | D_Typecode | F_Ptr)
```

These must also be manually added to h/rmacros.h.

Three of the *opt-return* type specification clauses are useful for implementing value types (the fourth is used for variable types; see below). These clauses add return/suspend statements to RTL of the form

```
return    type–name(expr)
suspend   type–name(expr)
```

type-name is the identifier naming the type. It determines the D_ constant used for the dword of the operation's result descriptor. *expr* is a C expression whose value is placed in the vword of the result. The particular *opt-return* clause chosen determines how the C value is stored in the vword. The clauses are

```
return C_integer
return block_pointer
return char_pointer
```

C_integer indicates that the value is cast to a C integer; see the definition of word in h/typedefs.h for the exact C type used. block_pointer indicates that the value is cast to (union block *); this is usually used for pointers to blocks in the block region. char_pointer indicates that the value is cast to (char *). Note, only descriptors of a special form may point into the string region; the storage used with return char_pointer must reside elsewhere.

As an example, the type specification for the cset type is

```
cset{c}: simple
         return block_pointer
```

Suppose a variable cp within an Icon operation written in RTL points to a cset block. Then the statement

```
return cset(cp);
```

constructs a result descriptor for the cset and returns it.

For a type with an associated block, a declaration for the block structure must be added to h/rstructs.h. By convention, the structure name is created by prepending b_ to the type name. The first word of a block must contain its T_ type code. If different instances of the block may vary in size, the second word of the block must contain this size in bytes. The structure name of the new block must be added to the union block declaration in h/rstructs.h. An allocation routine for the block must be added to runtime/ralc.r. The macros AlcFixBlk() and AlcVarBlk() are

useful in such routines; see other allocation routines for guidelines.

There are five arrays in runtime/rmemmgt.r that must be updated for all types. These are used by garbage collection and diagnostic routines. The array bsizes contains the sizes of the blocks for corresponding T_ type codes. An entry of −1 indicates a type for which there is no block. An entry of 0 indicates a block whose second word contains the size.

This section assumes that values of simple types implemented in the block region consist of single blocks containing no descriptors and no pointers to other blocks. Therefore for simple types, the arrays firstd, firstp, and ptrno, contain 0 for types with blocks and −1 for types with no blocks. More complicated implementations are discussed in the next section.

The array blkname contains strings used to identify types for use by debugging routines.

Storage for the values of a type usually should be allocated in the block region. However, for interfaces to packages written in C, it may be necessary to use storage that is not relocated by garbage collection. While it is possible to place types allocated with malloc() under control of garbage collection, this is complicated and beyond the scope of this appendix. See the implementation of co-expressions for an example of how this can be done. Alternatives are to ignore the storage leakage caused by unfreed storage or provide an Icon function, along the lines of close(), that explicitly frees storage associated with a value.

Three built-in functions must be updated to handle any new type. copy() and type() are in the file runtime/fmisc.r. image() is updated by changing the support routine getimage() in the file runtime/rmisc.r. If a type has a logical notion of size, then the size operator in runtime/omisc.r must be updated.

Several other support routines must also be updated. outimage() in the file runtime/rmisc.r produces the images of values for diagnostics and tracing. order() determines the collating sequence between types, anycmp() determines the collating sequence between any two values, and equiv() determines whether two values are equivalent (it is only updated for types, such as cset, for which a simple descriptor comparison is not adequate to determine equivalence). These routines are in the file runtime/rcomp.r.

At the end of this appendix is a check list of files that must be updated when a type is added to Icon.

Aggregate Types

Aggregate types have values with components that are other Icon values. The aggregate type specification provides more sophisticated RTL abstract type computations for the type. These in turn allow iconc to produce code that is better optimized.

For interpreter-only implementations, abstract type computations are not used and are optional in RTL code; the simple type specification may be used in that case. However, the discussion later in this section on block layout and on the storage management arrays still applies.

The *kind* clause of an aggregate type specification establishes and names abstract components for the type. The clause is of the form

kind ::= aggregate(*component*, ...)

component ::= *identifier* |
var *identifier opt–abrv*

Note, the *opt-return* clauses discussed in the previous section may be also used with aggregate types.

The aggregate specification can be thought of as establishing a sort of ‘‘abstract type record’’ whose fields, the abstract components, summarize the type information of the actual components of values in the type. Most types are give one abstract component. For example, the set type has the specification

```
set{S}: aggregate(set_elem)
        return block_pointer
```

where set_elem represents all the elements of a set.

Abstract components can be accessed using dot notation, and the new abstract type computation can be used to establish a new subtype of the type (subtypes only exist internally in the compiler and have no existence at the Icon language level). A subtype can be returned by the operation and has its own component types independent of

subtypes created elsewhere. The abstract type computation for Icon set intersection, the ** operator, uses both dot notation and a new expression. It performs intersection in the abstract type realm. x and y are the parameters of the operation and may contain different subtypes of the set type:

```
return new set(store[type(x).set_elem] ** store[type(y).set_elem])
```

(Note that the components can be thought of as references to information contained in a *type store* – thus the indexing notation.)

Components that represent Icon variables are preceded by var and may be given abbreviations for use in tracing type inferencing. For example, the list type has the specification

```
list{L}:  aggregate(var lst_elem{LE})
          return block_pointer
```

These components may be returned from operations and represent the component as a variable. For example, the abstract type computation for element generation operator when applied to a list is

```
return type(dx).lst_elem
```

where dx is the parameter of the operation. When a value rather than a variable is returned, the component must be ''dereferenced'' by indexing into the store, as in the abstract type computations of get():

```
return store[type(x).lst_elem]
```

Non-variable components must always be dereferenced.

For types, such as tables, that contain Icon values serving different purposes, it may be effective to establish several abstract components.

Aggregate types are implemented using blocks that contain descriptors, and they may be implemented using several kinds of blocks, with some blocks having pointers to others. When there are multiple blocks, there is always a *header* block that uses the T_ code of the type. Other blocks are given internal type codes; these codes must be added to h/rmacros.h and entries must be made in the storage management arrays.

Any descriptors in a block must be at the end. The type's entry in the firstd array is the location of the first descriptor. Any block pointers in the block must be contiguous. The type's entry in the firstp array is the location of the first pointer and its entry in the ptrno array is the number of pointers.

Keyword Variable Types

Keyword variable types have a type specification with a *kind* clause of the form

kind ::= variable *var–type–spec*

var–type–spec ::= initially *type* |
always *type*

type ::= *type–name* |
type ++ *type–name*

type–name ::= *identifier*

The compiler must be able to infer the types of values stored in a keyword variable. The initially option causes the keyword variable type to be treated as a set of global variables, each initialized to the given type specified by the *type* clause. The always option indicates that the keyword always contains values of the given type and the compiler does no actual inference on it. *type* may be the union of several types; this indicates that the type is uncertain and may be any of the ones specified. A special *type-name*, any_value, indicates complete uncertainty. The clause

```
always any_value
```

is a correct, although entirely imprecise, description of any keyword variable.

This appendix assumes that keyword variables are implemented by global descriptors (though other techniques are possible). The *opt-return* clause of the form

```
return descriptor_pointer
```

is useful for implementing keyword variables. The vword of a result descriptor from a corresponding return/suspend expression is of type struct descrip *.

Some of the same files must be updated for variable types as for value types. Type codes must be added to h/rmacros.h. The D_ code must have the F_Var flag set, for example:

```
#define D_Kywdint (T_Kywdint | D_Typecode | F_Ptr | F_Var)
```

The storage management tables and the outimage() routine also must be updated.

Other updates are unique to variable types. The global descriptor must be established. runtime/data.r contains its declaration, icon_init() in runtime/init.r initializes the descriptor, and h/rexterns.h contains an extern for it. Dereferencing must be updated; it is performed by deref() in runtime/cnv.r. Assignment must be updated; it is handled by the macro GeneralAsgn() in runtime/oasgn.r. The name() function is updated by changing the support routine get_name() in runtime/rdebug.r. The variable() function is updated by changing the support routine getvar() in runtime/rmisc.r.

The keyword itself goes in runtime/keyword.r. For example, &random is of type kywdint and is implemented by the descriptor kywd_ran; its definition is

```
keyword{1} random
   abstract {
      return kywdint
      }
   inline {
      return kywdint(&kywd_ran);
      }
end
```

For the interpreter, the keyword name must be added to icont/key_text.c. These names are in alphabetical order.

If the descriptor may contain a value under control of garbage collection, the support routine collect() in runtime/rmemmgt.r must be updated. postqual() preserves references to the string region; the macro Qual() is used to check for such references. markblock() preserves references to blocks; the macro Pointer() is used to check for such references.

The Complete Grammar for the Type Specification System

type–def ::= *identifier opt–abrv* : *kind opt–return*

kind ::= simple |
aggregate(*component*, ...) |
variable *var–type–spec*

component ::= *identifier* |
var *identifier opt–abrv*

var–type–spec ::= initially *type* |
always *type*

type ::= *type–name* |
type ++ *type–name*

type–name ::= *identifier*

opt–abrv ::= *nil* |
{ *identifier* }

```
opt–return      ::=  nil |
                     return block_pointer |
                     return descriptor_pointer |
                     return char_pointer |
                     return C_integer
```

A Check List for Adding Types to Icon

The following check list shows the files that must be updated when adding types to Icon. It is on a separate page for easy photocoping.

All Types

☐ common/typespec.txt – add type specification
☐ common/Makefile – uncomment entries near the end of the file
☐ h/rmacros.h – add T_*Type* macro
☐ h/rmacros.h – add D_*Type* macro
☐ runtime/rmemmgt.r – bsizes table
☐ runtime/rmemmgt.r – firstd table
☐ runtime/rmemmgt.r – firstp table
☐ runtime/rmemmgt.r – ptrno table
☐ runtime/rmemmgt.r – blkname table
☐ runtime/rmisc.r – update outimage()

All Value Types

☐ runtime/fmisc.r – update copy()
☐ runtime/fmisc.r – update type()
☐ runtime/rcomp.r – update anycmp()
☐ runtime/rcomp.r – update order()
☐ runtime/rcomp.r – update equiv()
☐ runtime/rmisc.r – update getimage()

Types Implemented In The Block Region

☐ h/rstructs.h – add declaration for the block structure
☐ h/rstructs.h – update the union block declaration
☐ runtime/ralc.r – add an allocation routine

Types With Sizes

☐ runtime/omisc.r – update size operator

All Keyword Variable Types

☐ h/rexterns.h – extern for keyword descriptor
☐ runtime/cnv.r – update deref()
☐ runtime/data.r – declaration for keyword descriptor
☐ runtime/init.r – initialize keyword descriptor
☐ runtime/keyword.r – add keyword declaration
☐ runtime/oasgn.r – update GeneralAsgn() macro
☐ runtime/rdebug.r – update get_name()
☐ runtime/rmisc.r – update getvar()
☐ icont/key_text.c – add keyword name

Keyword Variables That Must Be Garbage Collected

☐ runtime/rmemmgt.r – update collect()