The Implementation of Data Structures in Version 5 of Icon*

Ralph E. Griswold

TR 85-8

April 10, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants MCS81-01916 and DCR-8401831.

The Implementation of Data Structures in Version 5 of Icon

1. Introduction

Icon supports a variety of data types, some of which are unusual [1]. There are sophisticated access mechanisms for structures, including associative look-up and lists that can grow and shrink. Variables in Icon are untyped and structures can be heterogeneous, having elements of different types. For example, an element of a list can be a list — even itself. Storage management is automatic. There are no storage declarations; objects are created during program execution, space is allocated implicitly as it is needed, and space is reclaimed automatically.

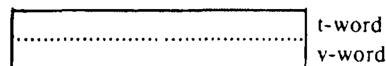
These features of Icon have a profound effect on the way that it is implemented. This report describes the implementation of data structures in Icon in some detail, but it does not attempt to justify all design decisions or to explain why every given implementation choice is better than an alternative. In fact, some aspects of the implementation were affected by earlier work whose traces are no longer evident.

This report describes data structures as they are implemented in Version 5.9 of Icon [2]; there are some implementation differences from earlier versions. The reader should be familiar with Icon, particularly its data structures. This report can be read alone or in conjunction with listings of the source code to illuminate details. The Icon "tour" [3] is a useful auxiliary reference. Some knowledge of C [4], in which most of Icon is written, also may be useful.

2. Descriptors

All values and variables in Icon are represented by *descriptors*. Descriptors are fixed in size for any given implementation. A descriptor either may contain a value (as in the case of an Icon integer) or it may contain a pointer to an object that contains the value (as in the case of an Icon real number).

The layout of data within a descriptor depends somewhat on the kind of value or variable that it represents. A descriptor normally consists of two words (C *ints*):



Descriptor Layout

The t-word (for "type") typically contains a type code and flags that indicate properties of the descriptor. The v-word (for "value") typically contains a value or a pointer to data representing the value. The dotted line between the two words of a descriptor is provided for readability and does not represent any real demarcation in memory.

There are 11 source-language data types in Icon* as well as records (which collectively constitute a single type for implementation purposes). There are also a number of internal data types, such as table elements, that are on a par with source-language data types, but which are not directly accessible from a source-language program.

There are two forms of variables, *ordinary* and *trapped*. An ordinary variable points to a descriptor that contains the corresponding value. A trapped variable is used in situations in which processing the variable requires some special action, such as in assignment to a table reference or to &pos. Some trapped variables point to blocks of data that contain information that is needed when the variable is accessed. Other trapped

*This includes the set data type that is available as a optional extension in Version 5.9 of Icon [2].

variables simply contain a code that indicates the action that is required. See Sections 4, 5, and 10.

The descriptors for all of these cases fall into six categories:

the null value
strings
integers
other values
ordinary variables
trapped variables

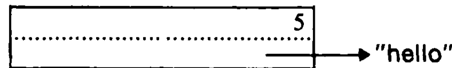
The null value has a unique representation in which both the t-word and v-word are zero:

The Descriptor for the Null Value

For strings, the t-word contains the length of the string (the number of characters in it) and the v-word is a pointer to the first character of the string:

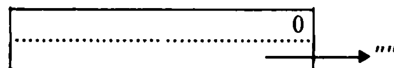
A Descriptor for a String

In order to make string descriptors more intelligible in the diagrams that follow, a pointer to a string is followed by the string in quotation marks rather than an address. For example, the string descriptor for "hello" is depicted as



A Descriptor for the String "hello"

Since the empty string has a length of zero, its v-word is always nonzero to avoid ambiguity with the null value. The actual value of the v-word of an empty string has no special significance. The empty string is indicated as follows:



An Empty String

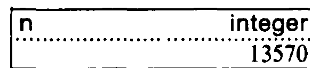
All descriptors except those for the null value and strings contain flags in the high-order bits of their t-words. In subsequent diagrams, these flags are represented symbolically as follows:

n descriptor is not the null value or a string
p v-word of the descriptor contains a pointer to nonstring data
v descriptor is a variable
t descriptor is a trapped variable
m descriptor is marked (used only during garbage collection)

Descriptors for values other than the null value and strings, as well as those for trapped variables, also contain a type code in the low-order bits of the t-word. There are 20 type codes represented by small integers. The actual values of the type codes are not important; they are represented symbolically as shown in the following list:

descriptor type	*type code*
integer	integer
long integer	long
real number	real
cset	cset
file	file
procedure	proc
list	lheader
set	sheader
table	theader
co-expression	estack
record	record
list element*	lelem
set element*	selem
table element*	telem
co-expression block*	eblock
substring trapped variable	tvsubs
table element trapped variable	tvtbl
&pos trapped variable	tvpos
&random trapped variable	tvrand
&trace trapped variable	tvtrac

For example, the Icon integer 13570 is represented by



Descriptor for the Integer 13570

Note that the n flag distinguishes this descriptor from a string whose first character might be at the address 13570 and whose length might have the same value as the type code for integer.

On the other hand, a list is represented by

Descriptor for a List

where the v-word points to the structure for the list. The p flag is used by the garbage collector.

3. Blocks

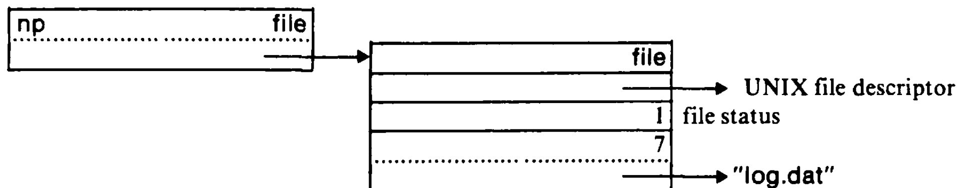
Most kinds of Icon data are represented by blocks of words. These blocks have a comparatively uniform structure that is designed to facilitate their processing during garbage collection.

The first word of every block contains a type code — the same code that is in the t-word of a descriptor that points to the block. Some blocks are fixed in size for all values of a given type. For example, all Icon values of type file point to blocks of the same size. For example, as a result of

```
in := open("log.dat")
```

the value of in is

*These are internal data types.



The Value of in

The string name of the file is used in tracing and diagnostic operations. Files are discussed in more detail in Section 12.

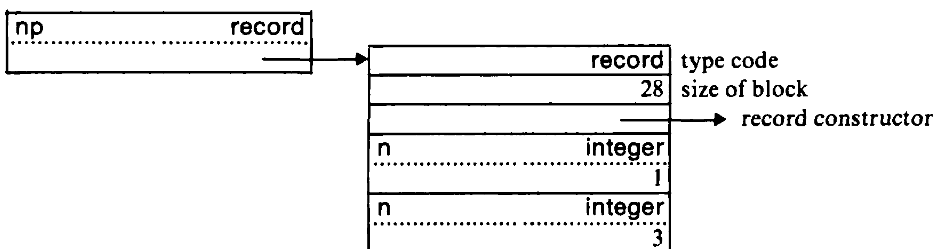
Blocks of some other types, such as record blocks, vary in size from value to value, but any one block is fixed in size and never grows or shrinks. If the type itself does not determine the size of the block, the second word in the block contains the size (in bytes). In the diagrams that follow, the sizes of blocks are given for 32-bit computers such as the VAX. For example, given the record declaration

```
record complex(r,i)
```

and

```
j := complex(1,3)
```

the value of j is:



A Record with Two Fields

The record constructor contains information that is needed to resolve field references. See Section 13.

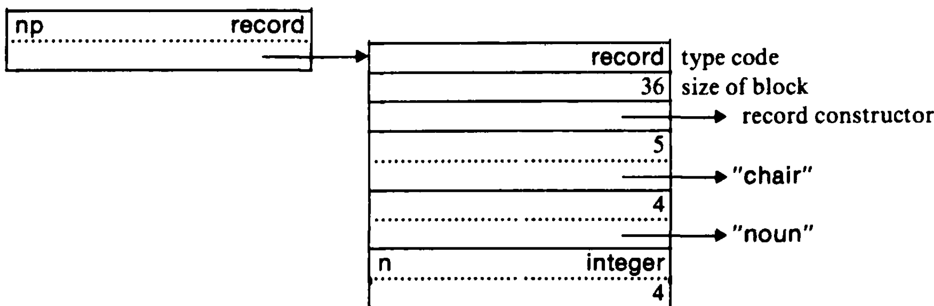
On the other hand, with the declaration

```
record term(value, code, count)
```

and

```
word := term("chair", "noun", 4)
```

the value of word is:



A Record with Three Fields

As illustrated by this example, blocks may contain descriptors as well as non-descriptor data. Non-descriptor data comes first in the block, followed by any descriptors, as illustrated by the preceding figure. The

location of the first descriptor in a block is constant for all blocks of a given type, which facilitates garbage collection.

Some blocks, such as those for the standard input and output files, are compiled as part of the Icon run-time system. Other blocks, such as those for the procedures declared in a program, are created when the program is translated, linked, and loaded into memory prior to execution. Such blocks are located in regions of memory that are static and they are not moved during program execution. Most blocks, such as those produced during execution by the creation of lists and other structures, are dynamically allocated in a "heap" region. Blocks in the heap region may be moved (relocated) during garbage collection.

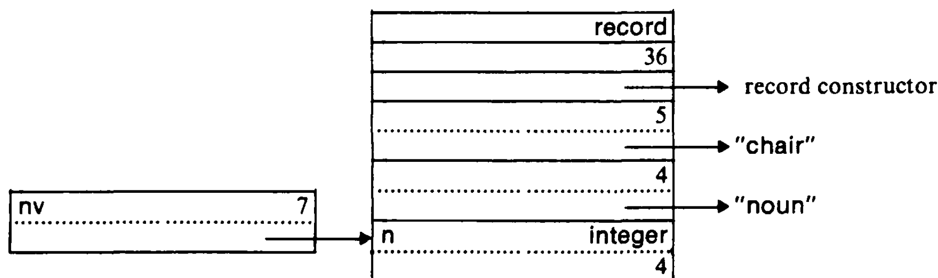
All pointers to blocks are in the v-words of descriptors, since descriptors contain the information needed to identify blocks (the p flag, the type code, and the address of the block). If a block in the heap is moved as a consequence of garbage collection, all pointers to it are changed accordingly.

4. Variables

Variables are represented by descriptors, just as values are. Variables for identifiers point to descriptors for the corresponding values. Ordinary variables always point to descriptors for values, never to other variables.

The values of local identifiers are kept on the stack, while the values of global and static identifiers are located at fixed places in memory. Variables that point to the values of identifiers are created on the stack by icode instructions [3] that correspond to the use of the identifiers in the program.

A variable that references a value in a data structure points directly to the descriptor for the value. The low-order portion of the t-word for such a variable contains the offset, in *words*, of the value descriptor from the top of the block in which the value is contained. This offset is used by the garbage collector. The use of words, rather than bytes, allows larger offsets. For example, the variable word.count for the record given in the preceding section is:



A Field Reference

Unlike an ordinary variable, a trapped variable does not point to a descriptor for its value, but instead either (1) points to a block that contains information that is necessary to dereference or assign to the variable, or (2) indicates a computation that must be performed to dereference or assign to the variable.

The first kind of trapped variable is used for substring and table references, which are described in Sections 5 and 10. The keyword &pos is an example of the second kind of trapped variable and is represented by

The Trapped Variable for &pos

The type code tvpos determines the computation to be performed. If &pos is dereferenced, as in

```
write(&pos)
```

the current position in the &subject, which is stored in a fixed location in the run-time system, is produced. On the other hand, in

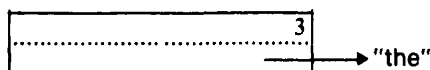
&pos := *expr*

the value of *expr* is stored at that location. Note, however, that this requires type checking and possible type conversion, since the value of *expr* may not be an integer. This is why an ordinary variable cannot be used for &pos.

5. Strings

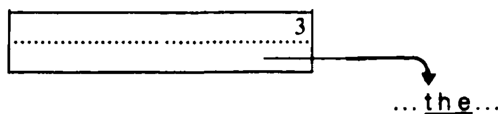
Icon strings, unlike C strings, are not null-terminated and may contain any of the 256 characters in the Icon character set. The characters of Icon strings, like C strings, are in consecutive locations (bytes) in memory.

As described earlier, an Icon string is represented by a descriptor that contains its length and the address of its first character. This address is a byte address and may not be at a word boundary. Such descriptors are called *string qualifiers* or simply *qualifiers*. The string "the" is represented by the qualifier



The Qualifier for "the"

The pointer to "the" is just a notational convenience. A more accurate representation is



The Qualifier for "the"

The actual value of the v-word might be 569a (hexadecimal), where the character t is at location 569a, the character h is at location 569b, and the character e is at location 569c.

String qualifiers are used to represent all Icon-style strings (as opposed to C-style strings that are used in the C computations). Because a qualifier delimits a string only by the address of its first character and its length, Icon strings can overlap or be contained within other Icon strings. Note that the computation of the length of an Icon string is simple and does not require examination of the characters in the string.

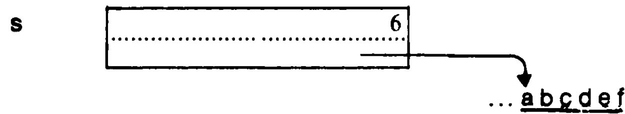
Like blocks, some strings are compiled into the run-time system and others, such as strings that appear as literals in a program, are loaded into memory along with the program. During program execution, Icon strings may be stored in work areas (usually referred to as "buffers"), but most newly created strings are allocated in a common string region. As source-language operations construct new strings, their characters are appended to the end of those already in the string region. The amount of space allocated in the string region typically increases during program execution until the region is full, in which point it is compacted by garbage collection, squeezing out characters that are no longer needed.

5.1 String Construction

Some operations that produce strings require the allocation of new strings, while other operations just produce new qualifiers. For example,

s := "ab" || "cdef"

allocates a string and produces a qualifier for it:



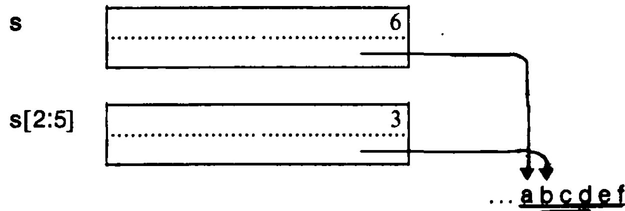
The Qualifier for a Newly Allocated String

This qualifier then becomes the value of s. The strings "ab" and "cdef" are literals and hence are loaded in with the program, instead of being in the string region.

On the other hand,

```
s[2:5]
```

does not allocate a string, but only produces a qualifier that points to a substring of s:



A Substring

Copies of them are made in the string region when the concatenation above is performed.

Operations such as concatenation allocate new strings and add them to the end of the current characters in the string region. At any time, the string region contains a sequence of consecutive characters into which various qualifiers point. In the case of a concatenation such as

```
s1 || s2
```

the string values of s1 and s2 are appended to the end of the existing string in the string region. One optimization is performed: In case s1 is the last string in the string region, s2 is simply appended to the end of the string region. Thus operations of the form

s := s || *expr*

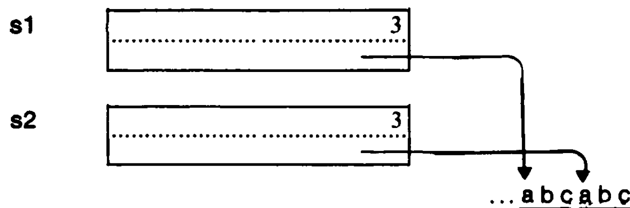
perform less allocation than operations of the form

s := *expr* || s

Except for this optimization, no string construction operation attempts to use another instance of a string that may occur somewhere else in the string region. Thus

```
s1 := "ab" || "c"
s2 := "a" || "bc"
```

produce two distinct strings:



An Example of Duplicate Allocation

5.2 Substrings

Assignment to a subscripted string, as in

```
s1[i] := s2
```

does *not*, in general, change the ith character of s1 to s2. Instead, it is equivalent to the following concatenation:

```
s1 := s1[1:i] || s2 || s1[i + 1:0]
```

When s1[i] is evaluated, the context (dereferencing or assignment) in which the result will be used may not be known. For example, if a procedure p contains the expression

```
return s[i]
```

and s is a global identifier, the use of the subscripted string depends on the context of the procedure invocation, which might be

```
write(p())
```

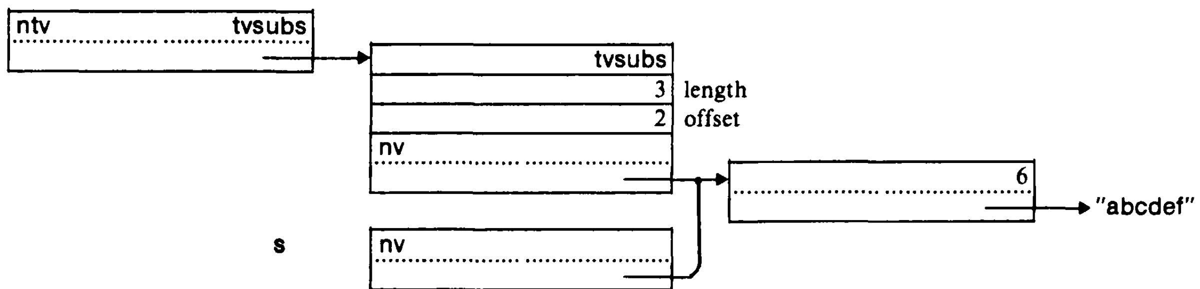
or

```
p() := ""
```

In order to handle such situations, when a string-valued variable is subscripted, a *substring trapped variable* is produced. For example, if

```
s := "abcdef"
```

the result of evaluating s[2:5] is a substring trapped variable that has the form



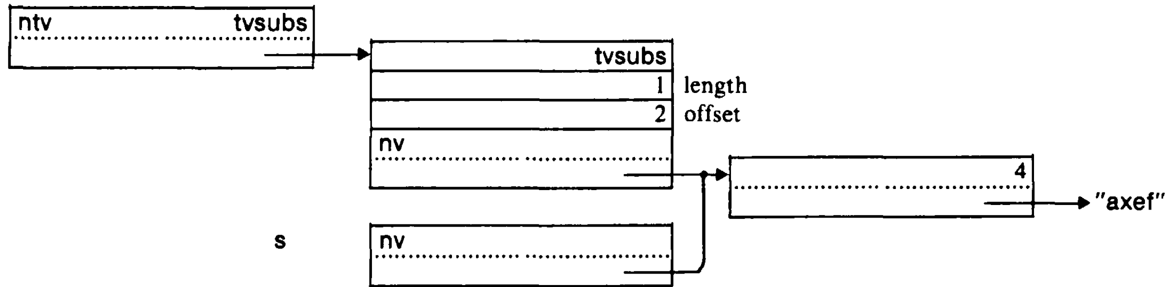
The Substring Trapped Variable Produced by s[2:5]

The length and offset of the substring provide the necessary information either to produce a qualifier for the substring in the case of dereferencing, or to construct a new string in the case of assignment. Note that the string itself is pointed to via a variable, since if an assignment is made, it is necessary to change the value of the variable that is subscripted.

After an assignment such as

```
s[2:5] := "x"
```

the situation is:



The Substring Trapped Variable After Assignment

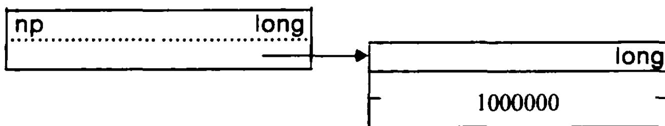
Note that the length of the subscripted portion of the string has be changed to correspond to the length of the string assigned to it. This reflects the fact that subscripting identifies the portions of the string before and after the subscripted portion ("a" and "ef", in this case). In the case of a multiple assignment to a subscripted string, only the original subscripted portion is changed. Thus, in

```
(s[2:5] := "x") := "yyyyy"
```

the final value of y is "ayyyyyef".

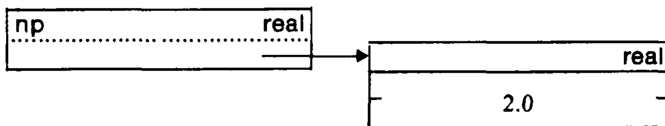
6. Numeric Types

Icon integers may be as large as C *longs*. On computers for which C *ints* are the same size as C *longs*, integers are contained in descriptors as illustrated by the diagrams above. On computers for which C *ints* are smaller than C *longs*, Icon integers that do not fit into C *ints* are contained in blocks that are pointed to by descriptors. For example, on the PDP-11 the integer 1000000 is represented by



The Integer 1000000 on the PDP-11

Icon real numbers are represented by C double-precision *floats* (64 bits) and are contained in blocks. For example, the real number 2.0 is represented by



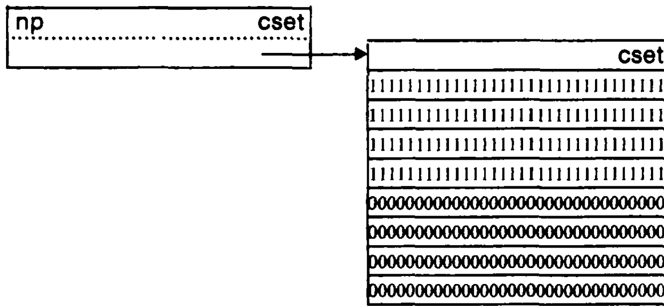
The Real Number 2.0

Thus a real number block is 5 words (10 bytes) on a 16-bit computer and 3 words (12 bytes) on a 32-bit computer.

7. Csets

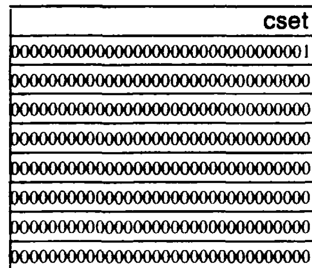
Since Icon uses 8-bit characters, regardless of the computer on which it is is implemented, there are 256 different characters that can occur in csets. A cset block consists of a header word containing the cset type code followed by 256 bits, each of which represents the presence or absence of a character in the cset. Thus a cset block, including the heading type word, is 17 words (34 bytes) long on a 16-bit computer and 9 words (36 bytes) long on a 32-bit computer.

A bit is 1 if the corresponding character is in the cset and 0 otherwise. The first 128 bits correspond to the ASCII character set, as illustrated by the value of **&ascii**:



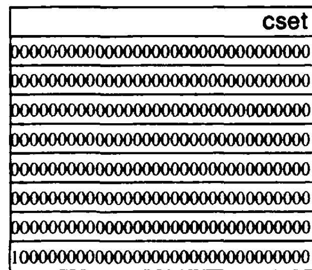
The Value of &ascii

Because of the way that memory is displayed in these diagrams, the rightmost bit in each word has the lowest address. Thus, the cset block for the first character, '\00', is



The Cset Block for '\00'

Similarly, the cset block for the last character, '\xff', is



The Cset Block for '\xff'

Another example is the cset for 'aeiou':

cset
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000001000001000001000100010
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000

The Cset Block for 'aeiou'

8. Lists

Lists can be accessed by position, but they also can grow and shrink at their ends as the result of stack and queue operations. The structures used for lists therefore are more complicated than those that would be needed to support only positional access.

A list consists of a *list header block*, which contains the usual type word, the current size of the list (the number of elements in it), and descriptors that point to the first and last blocks on a doubly-linked chain of *list element blocks* that contain the actual list elements.

When a list is created, either by

list(i, *expr*)

or by

$[expr_1, expr_2, \ldots, expr_n]$

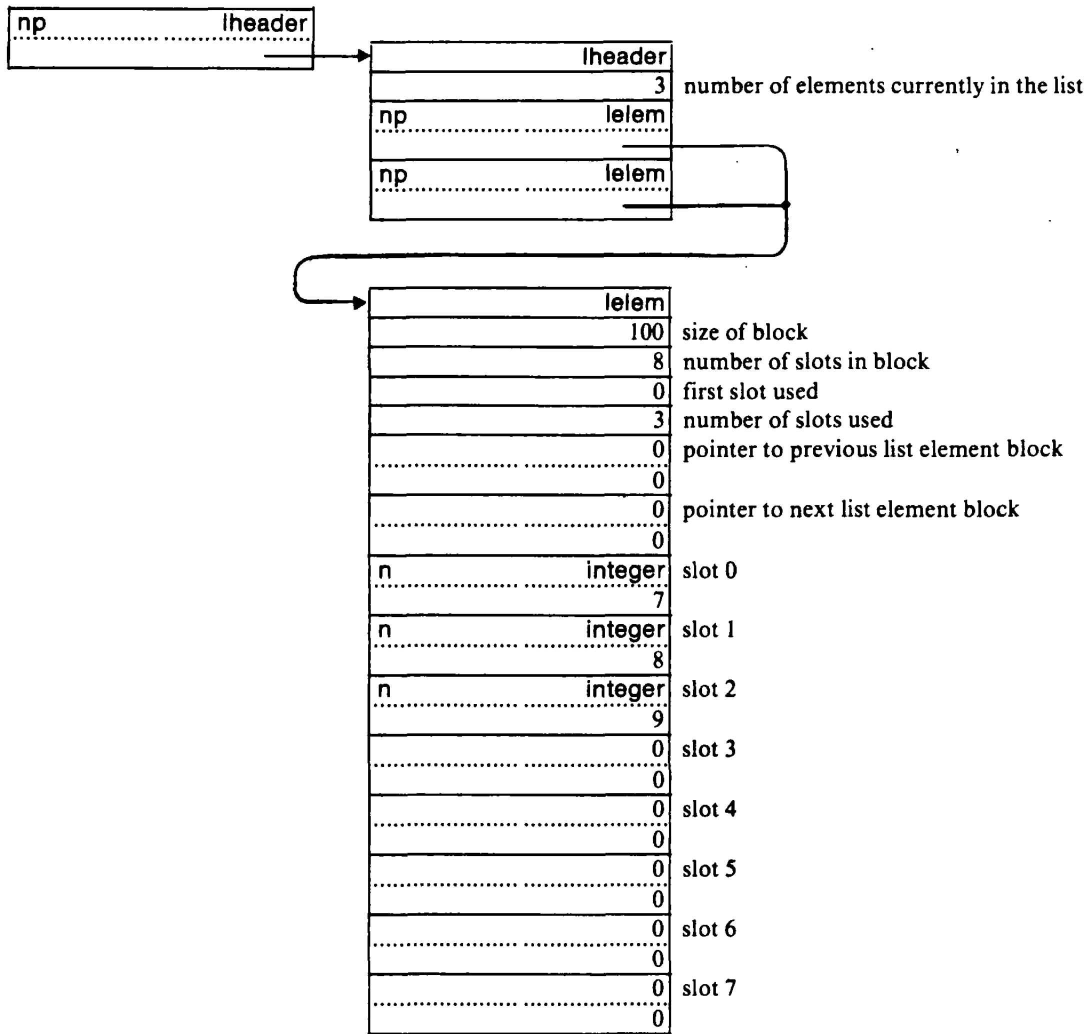
there is only one list element block. Other list element blocks may be added to the chain as the result of pushes or puts.

A list element block contains the usual type word, the size of the block (in bytes), three words used to determine the locations of elements in the list element block, and descriptors that point to the next and previous list element blocks, if any. (The null descriptor indicates the absence of a pointer.) Following this data are slots for elements (slots always contain valid descriptors, even if they are not used to hold list elements). The number of slots in a list element block is at least eight (an *ad hoc* implementation constant). This allows some expansion room for adding elements to lists, such as the empty list, that are small initially.

The data structures for a list are illustrated by the result of evaluating

```
a := [7, 8, 9]
```

which produces the following structures:



Data Structures for the List [7, 8, 9]

Note that there is only one list element block and that the slot indexing is zero-based. Unused slots contain null values that are logically inaccessible.

8.1 Storage of Elements in List Blocks

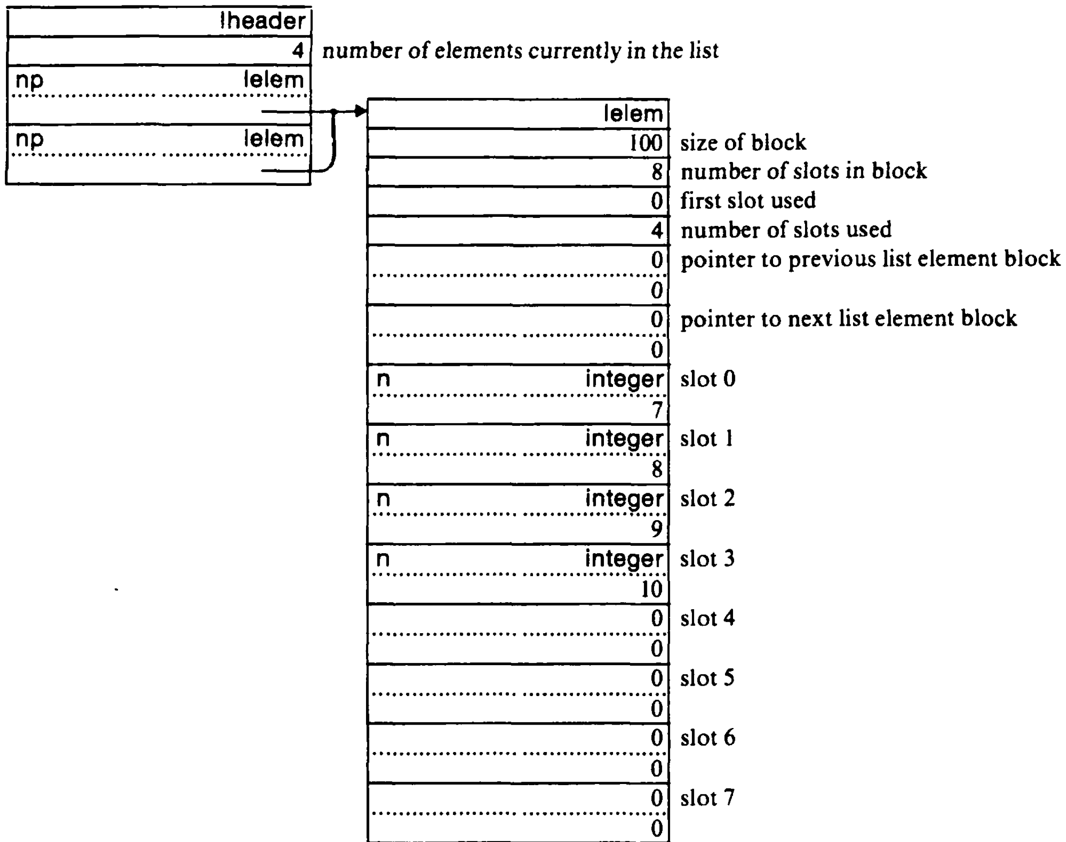
Elements in a list element block are stored as a circular queue. If an element is added to the end of a list, as in

```
put(a, 10)
```

the result is equivalent to

```
a := [7, 8, 9, 10]
```

The value is added to the "end" of the last list element block (assuming there is an unused slot). The result is:



The List Element Block After a put

Note that the increase in the number of elements in the list is reflected in the list header block.

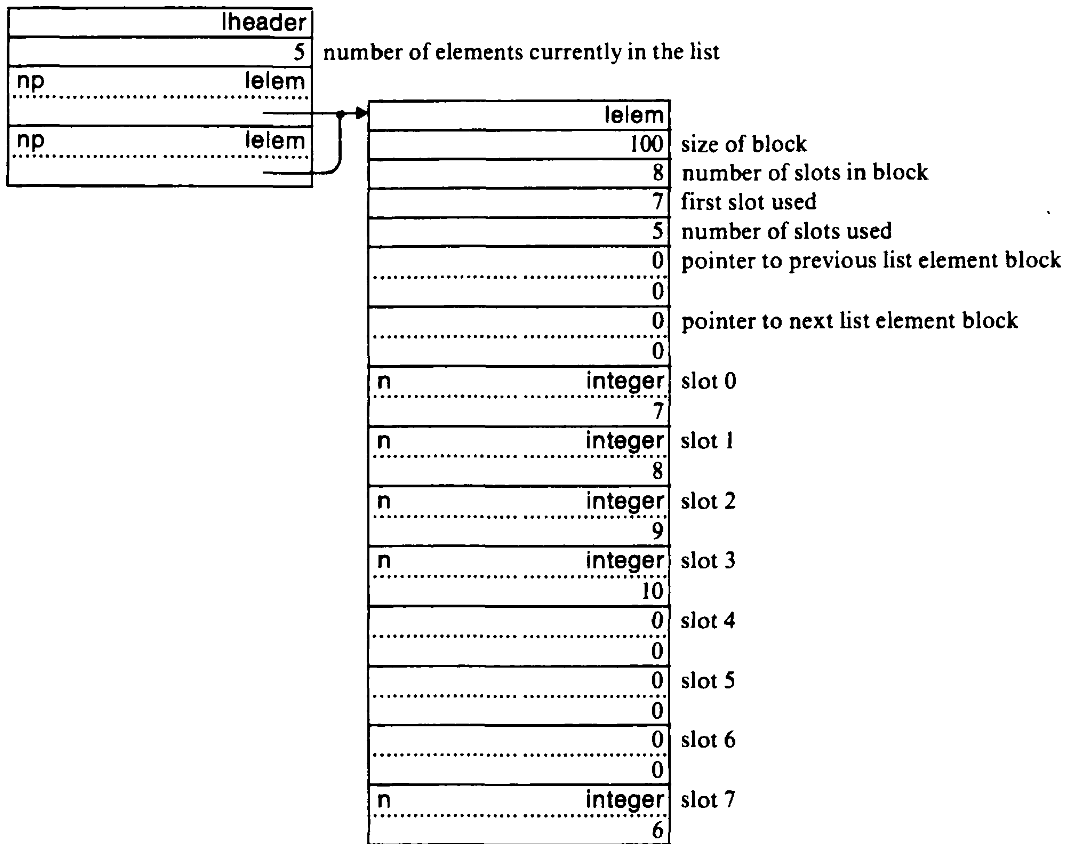
If an element is added to the beginning of a list, as in

```
push(a, 6)
```

the result is equivalent to

```
a := [6, 7, 8, 9, 10]
```

The new element is put at the "beginning" of the first list element block (assuming there is an unused slot). For the case above, the result is:



The List Element Block After a push

Note that the "beginning", which is before the first physical slot in the list element block, is the last physical slot. The locations of elements that are in a list element block are determined by the three integers at the head of the list element block — "removal" of an element by a pop, get, or pull does not shorten the list element block or overwrite the element — the element merely becomes inaccessible[1].

8.2 Adding and Removing List Blocks

If an element is added to a list and no more slots are available in the appropriate list element block, a new list element block (with eight slots) is allocated and linked in. For example.

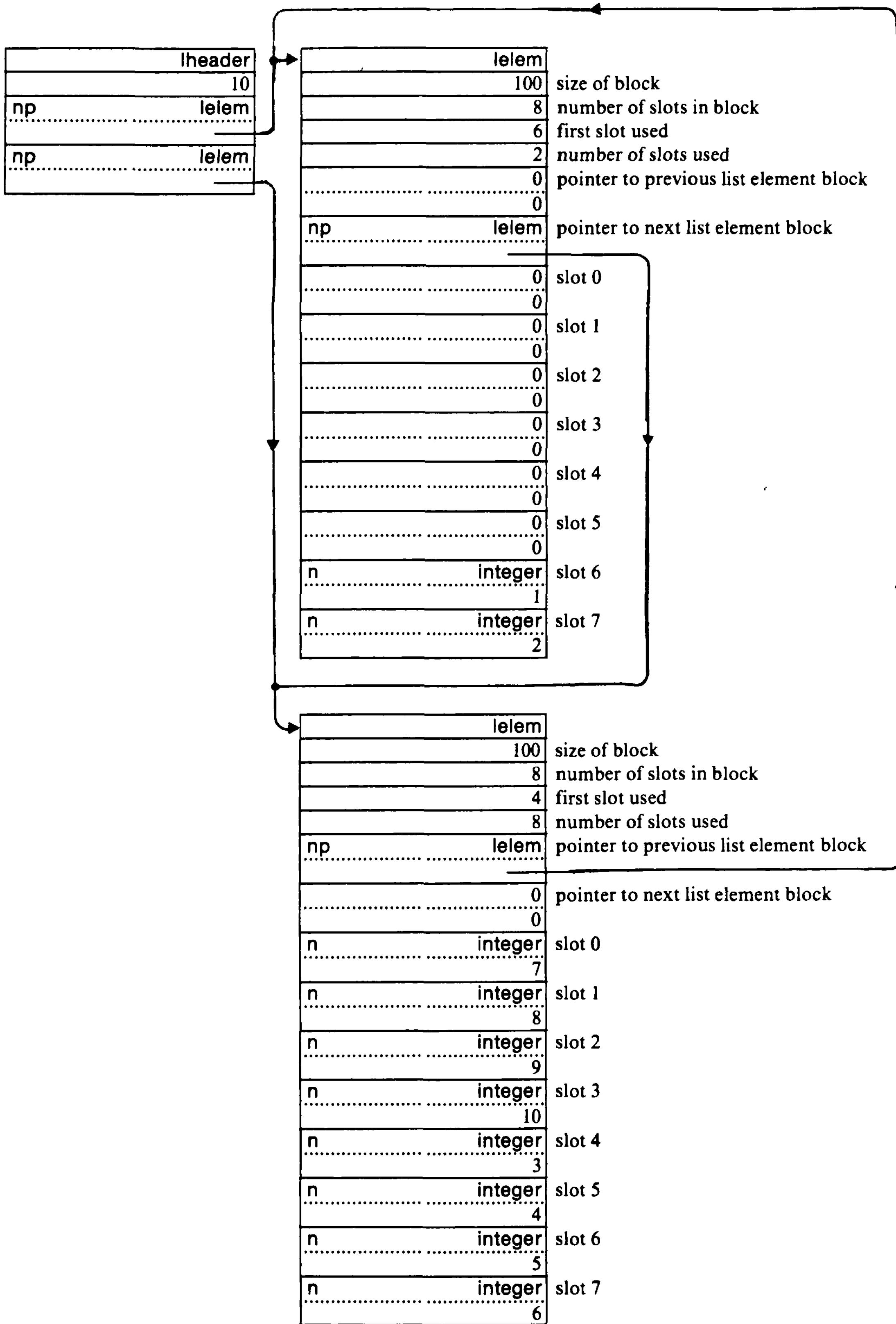
```
every push(a, 5 to 1 by -1)
```

is equivalent to

```
a := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The result is:

[1]Pathological source-language code can access such elements, at least transiently.



The Addition of a List Element Block

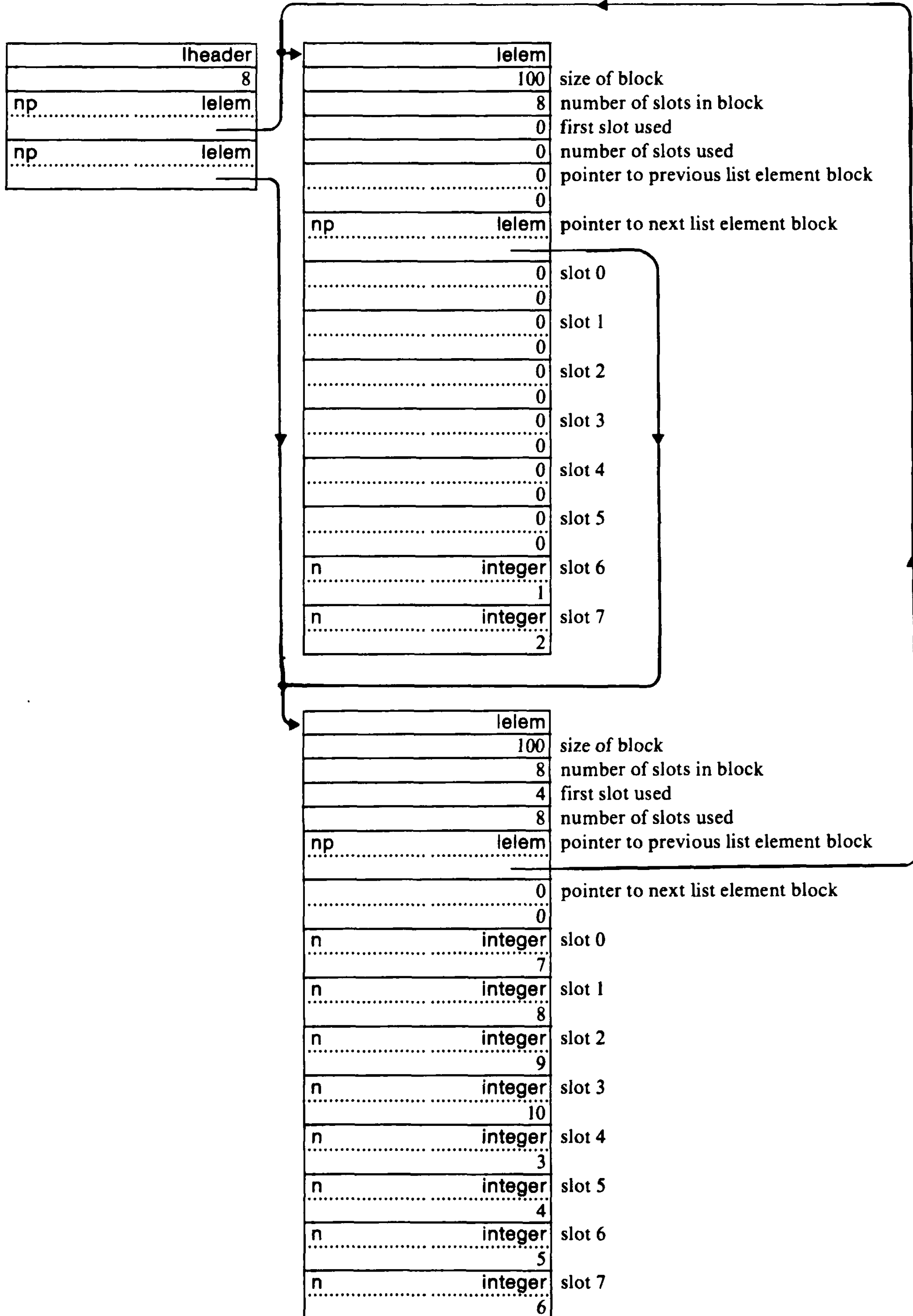
As elements are removed from a list by get (which is synonymous with pop) or pull, the indices in the appropriate list element block are adjusted. Thus if two elements are popped, as in

```
pop(a); pop(a)
```

the result is equivalent to

a := [3, 4, 5, 6, 7, 8, 9, 10]

and the structures become:



The Result of Removing Elements from a List Element Block

Note that the second list element block is still linked in the chain, even though it no longer contains any elements that are logically accessible. A list element block is not removed from the chain when it becomes empty, but only when an element is removed from a list that already has an empty list element block. Thus there is always at least one list element block on the chain, even if the list is empty. Aside from simplifying the access to list element blocks from the list header block, this avoids repeated allocation in the case that pop/push pairs occur at the boundary of two list element blocks.

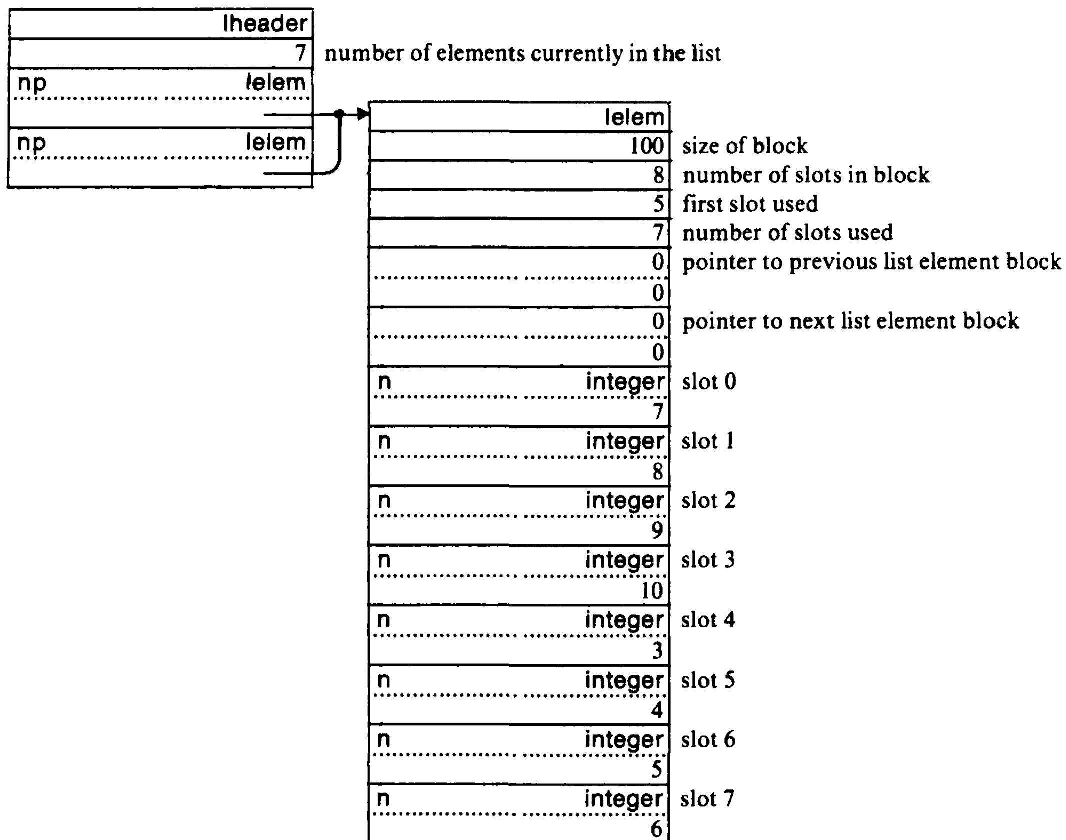
Continuing the example above,

```
pop(a)
```

which is equivalent to

```
a := [4, 5, 6, 7, 8, 9, 10]
```

causes the empty list element block to be removed:



The Removal of an Empty List Element Block

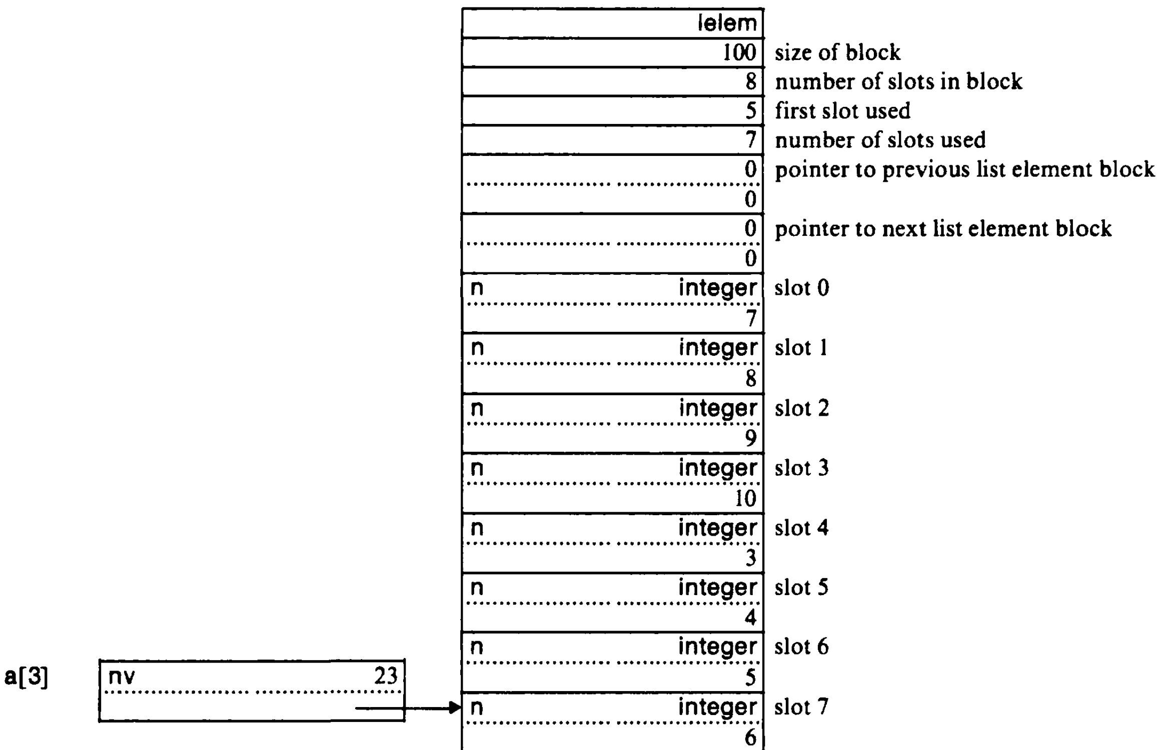
Note that the value 3 is still physically in the list element block, although it is logically inaccessible.

8.3 Positional References to Lists

A positional reference of the form a[i] requires locating the correct list element block (note that out-of-range references can be determined by examining the list header block). If the list has a number of list element blocks, this involves linking through the list element blocks, keeping track of the count of elements in each block until the appropriate one is reached. Nonpositive specifications are converted to positive ones before accessing the chain of list element blocks.

The result of evaluating a[i] is a variable that points to the appropriate slot in the list. For the preceding

example, a[3] produces a variable that points to the descriptor for the value 6 in [4, 5, 6, 7, 8, 9, 10].



Referencing a List Element

Note the offset of 23 words in the t-word of the variable.

8.4 Other List Construction Operations

The other sources of lists are list sectioning (a[i:j] and its variants), list concatenation, copy(a), sort(a), and sort(t).

In the case of a list section, a new list is constructed and the specified elements are copied into it. The new list has only one list element block, regardless of the number of list element blocks in the list in which the section is specified. As with explicit list creation, at least eight slots are provided. The function copy(a) is similar — one list element block is created and all the elements from a are copied into it.

In list concatenation, two list element blocks are created, one for each of the lists specified in the concatenation (regardless of their list element block structure). These two list element blocks are linked together after the elements are copied into them. The use of two list element blocks rather than a single large one is primarily a matter of convenience.

The function sort(a) is very similar to copy(a), but sort(t) is more interesting, since a list of two-element lists is produced. These two-element lists have list element blocks with only two slots. This is the only case in which list element blocks have less than eight elements.

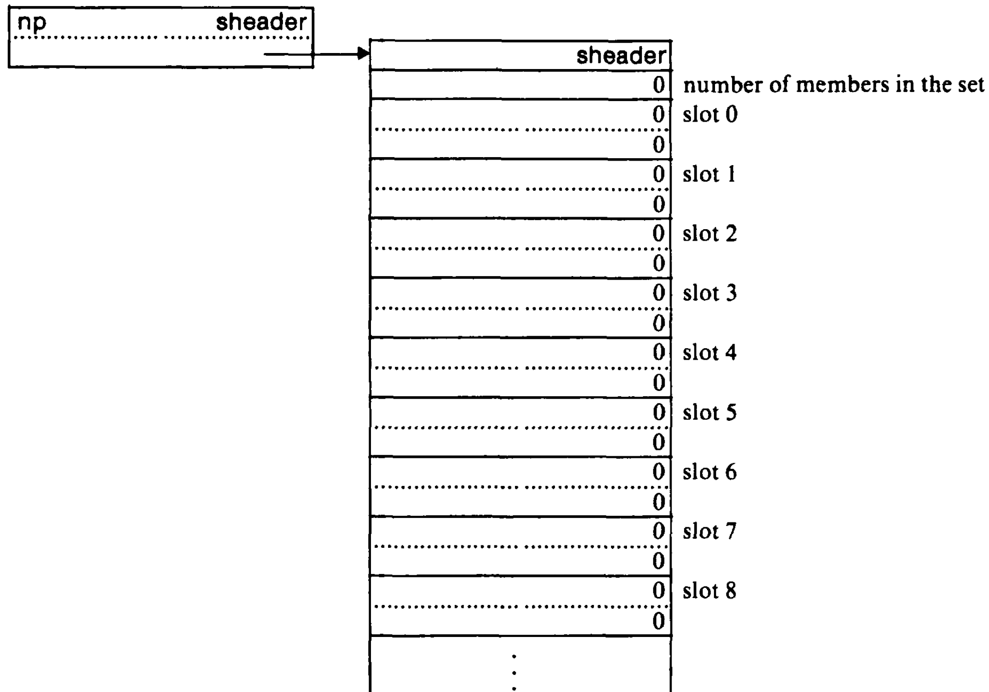
9. Sets

Since sets can contain an arbitrary number of members of any type, a hash lookup scheme is used to provide an efficient way of locating set members. For every set there is a set header block that contains a word for the number of members in the set and slots that serve as heads for (possibly empty) linked lists ("buckets") of set element blocks. The number of slots is an implementation parameter. There are 37 slots in table header blocks on the VAX, but only 13 slots on the PDP-11, where the address space is more limited.

The structure for an empty set, produced by

s := set([])

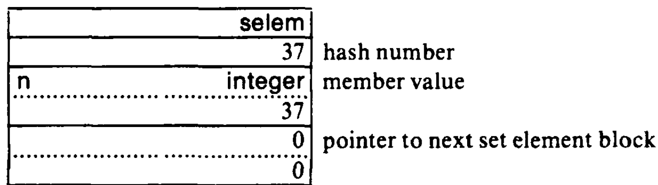
is:



An Empty Set

Each member of a set is contained in a separate set element block. When a value is a looked up in a set (for example, to add a new member), a hash number is computed from this value. The hash number modulo the number of slots is used to select a slot.

Each set element block contains a descriptor for its value, the corresponding hash number, and a pointer to the next set element block, if any, on the list. For example, the set element block for the integer 37 is:



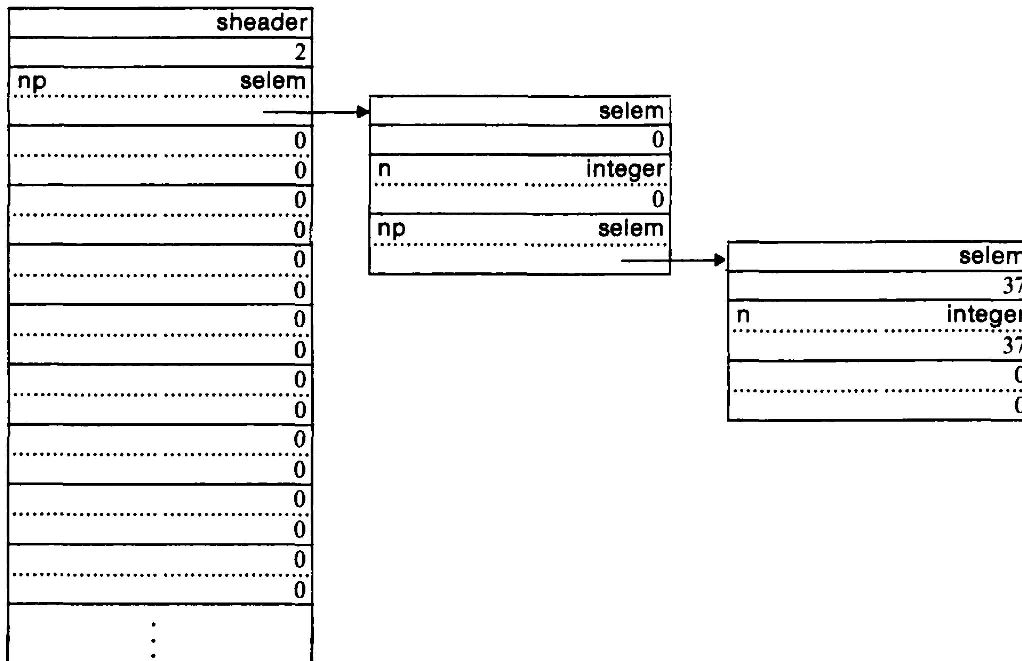
The Set Element Block for the Integer 37

As illustrated by this figure, the hash number for an integer is just the value of the integer. This member goes in slot 0 on the VAX, since its remainder on division by the number of slots is zero. See Section 11 for a general discussion of hash computations.

The structures for the set

s := set([37, 0])

are:



A Set with Two Members

This example was chosen for illustration, since both 0 and 37 go in slot 0.

In searching the list, the hash number of the look-up value is compared with the hash numbers in the set element blocks. If a match is found, the value in the set element block may or may not be the same as the value being looked up (collisions in the hash computation are inevitable). Thus if the hash numbers are the same, it is necessary to compare the values. The comparison that is used corresponds to the source-language operation

```
x === y
```

To improve the performance of the look-up process, the set elements in each linked list are ordered by their hash numbers. When a chain of set element blocks is examined, the search stops if a hash number of an element on the chain is greater than the hash number of the value being looked up.

If the value is not found and the lookup is being performed to insert a new member, a set element block for the new member is created and linked into the list. The word in the set header block that contains the number of members is incremented to reflect the insertion. For example,

```
insert(s, -37)
```

inserts a set element block for –37 at the head of the list in slot 0, since its hash value is –37.

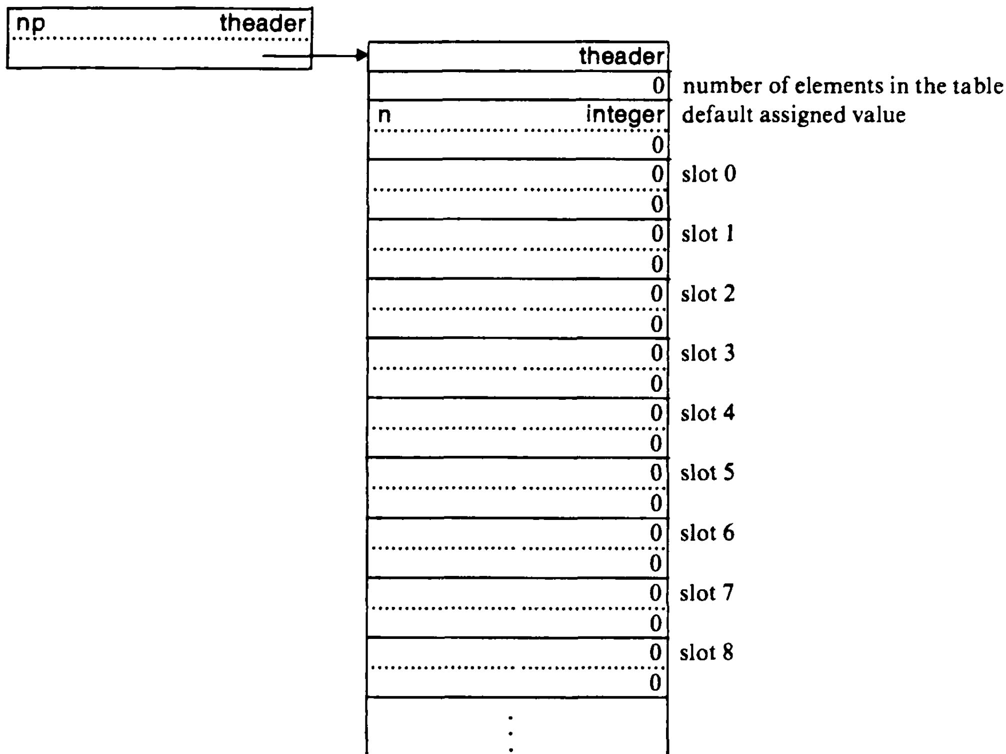
10. Tables

Tables are implemented in a fashion similar to sets, with a header block containing slots for elements ordered by hash numbers. A table header block contains an extra descriptor for the default assigned value.

An empty table with the default entry value 0, produced by

```
t := table(0)
```

illustrates the structure of the table header block:

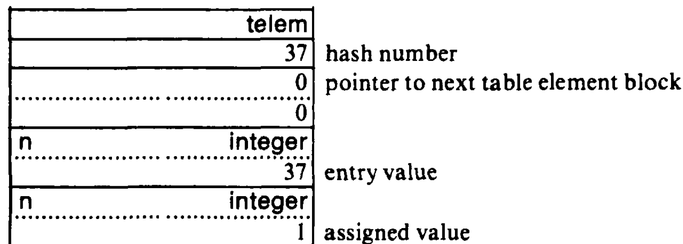


An Empty Table

Table lookup is more complicated that set lookup, since table elements contain both an entry value and an assigned value and a new table element can be created by assignment to a table reference with an entry value that is not in the table.

The structure of a table element block is illustrated by the one produced for an assignment such as

```
t[37] := 1
```



The Structure of a Table Element Block

In the case of a table reference such as

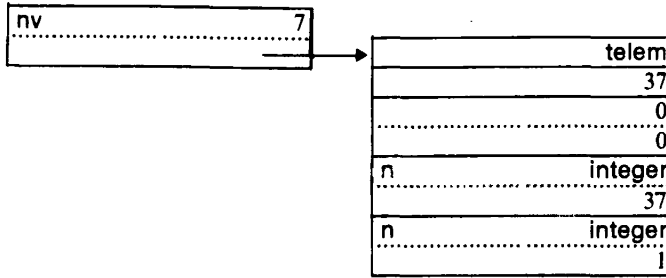
```
t[x]
```

the hash number for the entry value x is used to select a slot and the corresponding list is searched for a table element block that contains the same entry value. As for sets, comparison is first made using hash numbers; values are compared only if their hash numbers are the same.

If a table element block with a matching entry value is found, a variable that points to the corresponding assigned value is produced. For example, if 37 is in t as illustrated above, a subsequent reference

```
t[37]
```

produces:



Result of a Table Reference

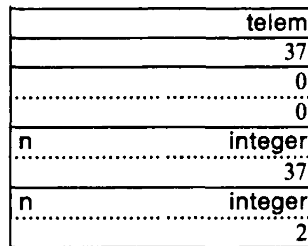
If this variable is dereferenced, as in

```
write(t[37])
```

the value 1 is written. On the other hand, if an assignment is made to this variable, as in

```
t[37] +:= 1
```

the assigned value in the table element block is changed:

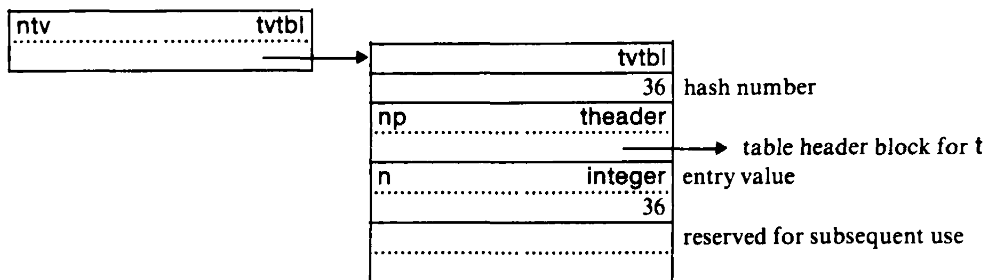


Result of Assignment to a Table Reference

If a table element with a matching entry value is not found, a table element trapped variable is created. Suppose, for example, that the entry value 36 is not in the table t. Then

```
t[36]
```

produces the following result:



A Table Element Trapped Variable

The last descriptor in the table element trapped variable block is reserved for subsequent use, as described below.

The use of this trapped variable depends on the context in which the reference that produced it is used. If it is dereferenced, as in

```
write(t[36])
```

the default assigned value, 0, which is in the table header block for t, is produced. It is possible, however, for elements to be inserted in a table between the time the table element trapped variable is created and the time it is dereferenced. An example is

```
write(t[36], t[36] := 2)
```

Since functions do not dereference their arguments until all the arguments have been evaluated, the result of dereferencing the first argument of write should be 2, not 0. In order to handle such cases, when a table element trapped variable is dereferenced, its list must be searched again to determine whether to return the assigned value of a newly inserted element or to return the default value.

If an assignment is made to the table reference, as in

```
t[36] +:= 1
```

the table element trapped variable is converted to a table element and inserted in the appropriate place in the table. Note that the structures of table element blocks and table element trapped variable blocks are the same, allowing this conversion without allocating a new table element block.

It is necessary to search the list for its slot again to determine the place to insert it. As in the case of dereferencing, elements may have been inserted in the table between the time that the table element trapped variable is created and the time that a value is assigned to it.

Normally, no matching entry is found and the table element trapped variable, transformed into a table element block, is inserted with the new assigned value. If a matching entry is found, its assigned value is simply changed.

Note that reference to a value that is not in a table requires only one computation of the hash value, but two lookups in the list of table element blocks for its slot.

11. Hash Computations

The purpose of using hash numbers is to reduce the time necessary to determine if a value is in a set or table. Storing the hash numbers in element blocks speeds the search, since comparison of hash numbers avoids the more time-consuming comparison of values in most cases.

The computation of hash numbers is an interesting problem, since sets and tables can contain values of any type. As shown above, the computation of the hash number for an integer is trivial. In the case of strings, the hash number is determined by adding the numerical values of the characters of the string, up to a maximum of 10. The length of the string is added to the result. For csets, the hash number is the exclusive-or of the words containing the bits for the cset.

Structures such as lists and records pose a conceptual problem, since the hash number computation for a value must produce the same number regardless of when the computation is made, and hence must be based on some unchangeable attribute of the value. However, the elements in a list may change, as well as its size. Even the location of a list (the address of its list header block) may change because of garbage collection [3]. In fact, the only unchangeable attribute of a list is its type, lheader. The situation is the same for sets and tables. In these cases, the types are used as hash numbers. This differentiates lists, sets, and tables, but all values of a given structure type have the same hash number. For example, all lists go into the same slot and their elements all have the same hash number.

Consequently, lookup for structure types is linear. Fortunately, few programs construct sets or tables that contain a large number of elements that are structures of the same type. For such situations, better performance could be obtained if there were a word for a hash number (such as the time of creation) in each structure header block. This would increase the size of every structure, however.

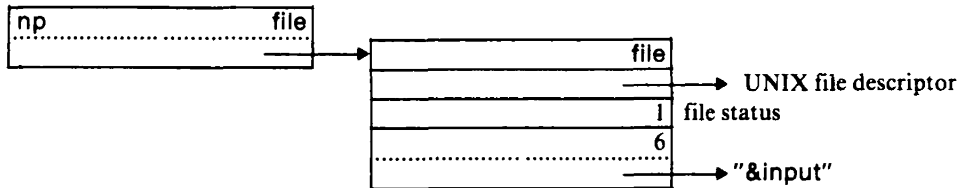
12. Files

A value of type file in Icon points to a block that contains the UNIX* file descriptor, a status word, and the string name of the file. The file status values are

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

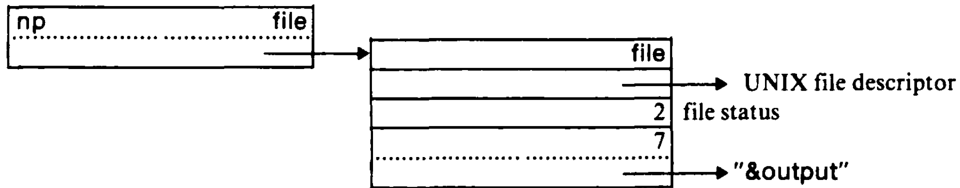
0	closed
1	open for reading
2	open for writing
4	open to create
8	open to append
16	open as a pipe

For example, the value of &input is



The Value of &input

while the value of &output is

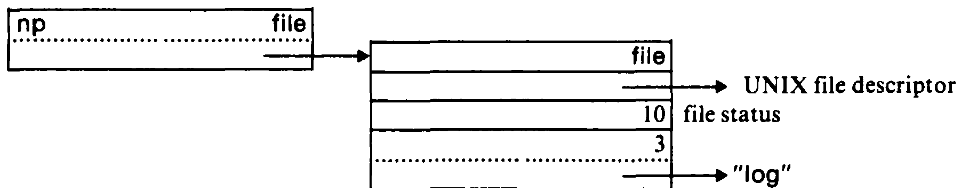


The Value of &output

Another example is

```
out := open("log", "a")
```

for which the value of out is



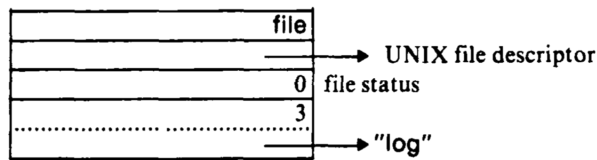
The Value of out

Note that the file status is 10, corresponding to being open for writing and appending.

Closing a file, as in

```
close(out)
```

merely changes its file status:



A Closed File

13. Procedures

There are three kinds of procedures in Icon: declared procedures, built-in functions, and record constructors. All are source-language values. The blocks for the three kinds of procedures are similar, differing only in detail.

Declared Procedures

For declared procedures, the procedure block contains the usual type and block size words, followed by five words that characterize the procedure:

address for the procedure in the interpretable program (icode)
the number of arguments (formal parameters) in the procedure
the number of dynamic local identifiers in the procedure
the number of static local identifiers in the procedure
the index of the first static local identifier in the procedure

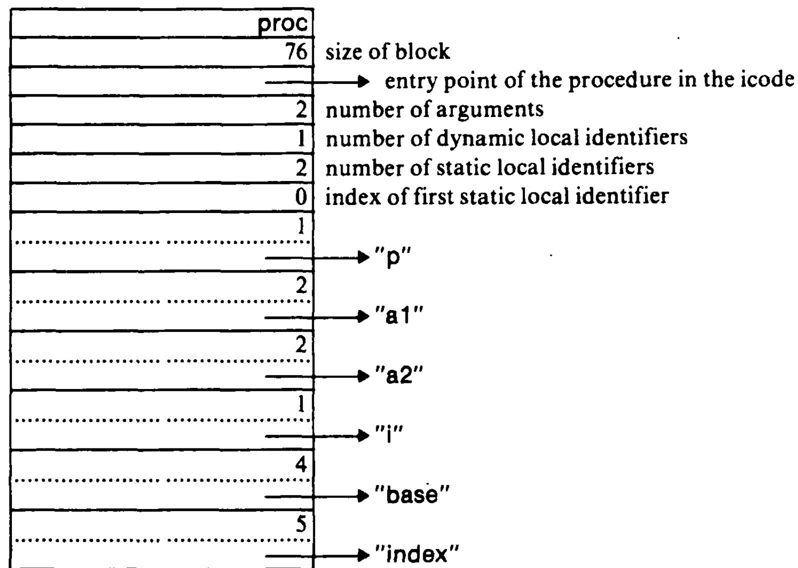
The remainder of the procedure block contains descriptors: first the string name of the procedure and then the string names of the arguments, dynamic local identifiers, and static local identifiers, in that order. The procedure name is needed for tracing. The function display(f, i) uses the index of the first static local identifier, the procedure name, and the names of the local identifiers.

Descriptors for the values of arguments and dynamic local identifiers are allocated on the stack when a procedure is called. The values of static local identifiers for all procedures are contained in a single array at a fixed location in memory.

For example, the procedure declaration

```
procedure p(a1, a2)
   local i
   static base, index
      .
      .
      .
end
```

has the following procedure block:

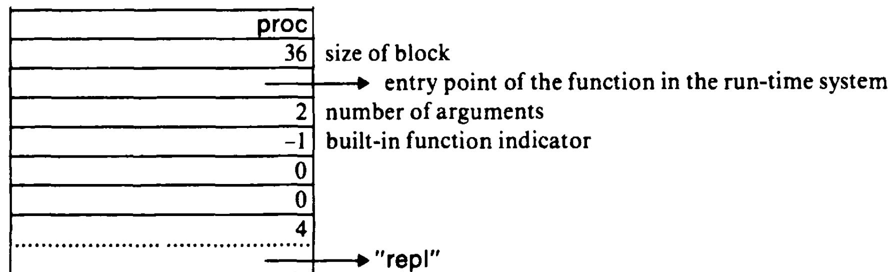


A Procedure Block

The 0 value for the index in the static identifier region indicates that base is the first static local identifier in the program (the indices of static identifiers are zero-based).

Built-In Functions

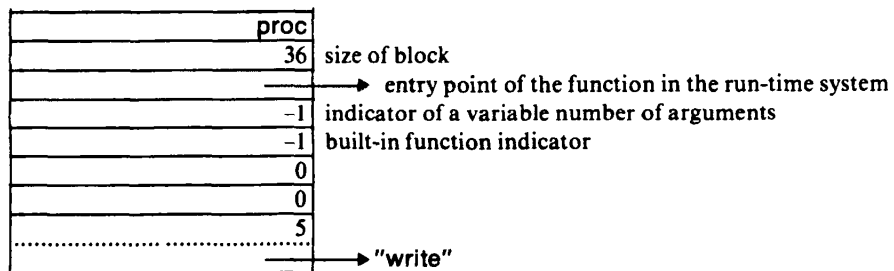
Functions are distinguished from declared procedures by the value -1 in the word that otherwise contains the number of dynamic local identifiers. An example is



The Procedure Block for the Function repl

The entry point refers to the corresponding C function in the run-time system. Note that there are no argument names.

Some functions, such as write, allow an arbitrary number of arguments. This is indicated by the value -1 in place of the number of arguments:



The Procedure Block for the Function write

Record Constructors

A record declaration produces a record constructor, which is analogous to a built-in function. For example,

```
record term(value, code, count)
```

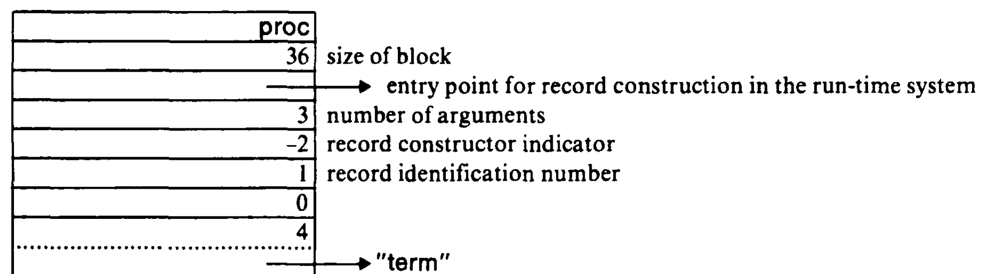
produces a record constructor term that creates records of type term, as in

```
x := term("noun", 5, 0)
```

Record constructors are distinguished from declared procedures and built-in functions by the value -2 in place of the number of dynamic local identifiers. The entry point refers to the C function that constructs record blocks. Each record declaration is distinguished by a unique record identification number. This number appears in place of the number of static local identifiers. For example, if the first record declaration in a program is

```
record term(value, code, count)
```

then its record identification number is 1 and the procedure block for its record constructor is:



The Procedure Block for the Record Constructor term

The record identification number could be used to distinguish records of different types for the purpose of hashing, but it is not.

14. Co-Expressions

A co-expression consists of a stack on which the evaluation of the co-expression takes place, a descriptor that points to the most recent activator of the co-expression, pointers related to expression evaluation, a *refresh block*, and a word containing the number of times the co-expression has been activated.

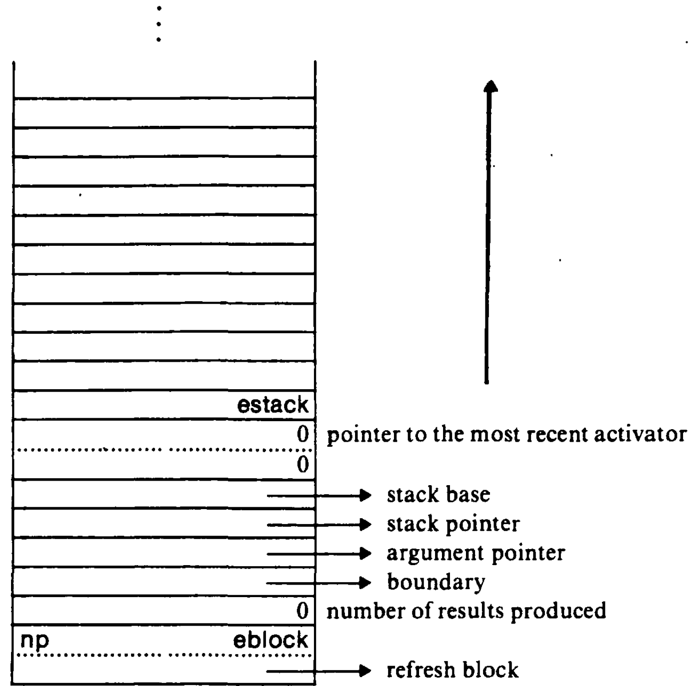
Consider the following procedure

```
procedure labgen(tag, i, j)
   return create tag || (i to j)
end
```

and the co-expression produced by

```
e := labgen("L", 1, 1000)
```

The value assigned to e is:



A Co-Expression Stack

On the VAX, the stack grows from large addresses toward smaller ones, so when the co-expression is activated the stack grows away from the type word.

Prior to the first activation of e, the most recent activator is null. The stack base, stack pointer, argument pointer, and boundary refer to stack locations that are related to expression evaluation. The data pushed on the stack when an expression is evaluated is machine dependent. See [5] for details.

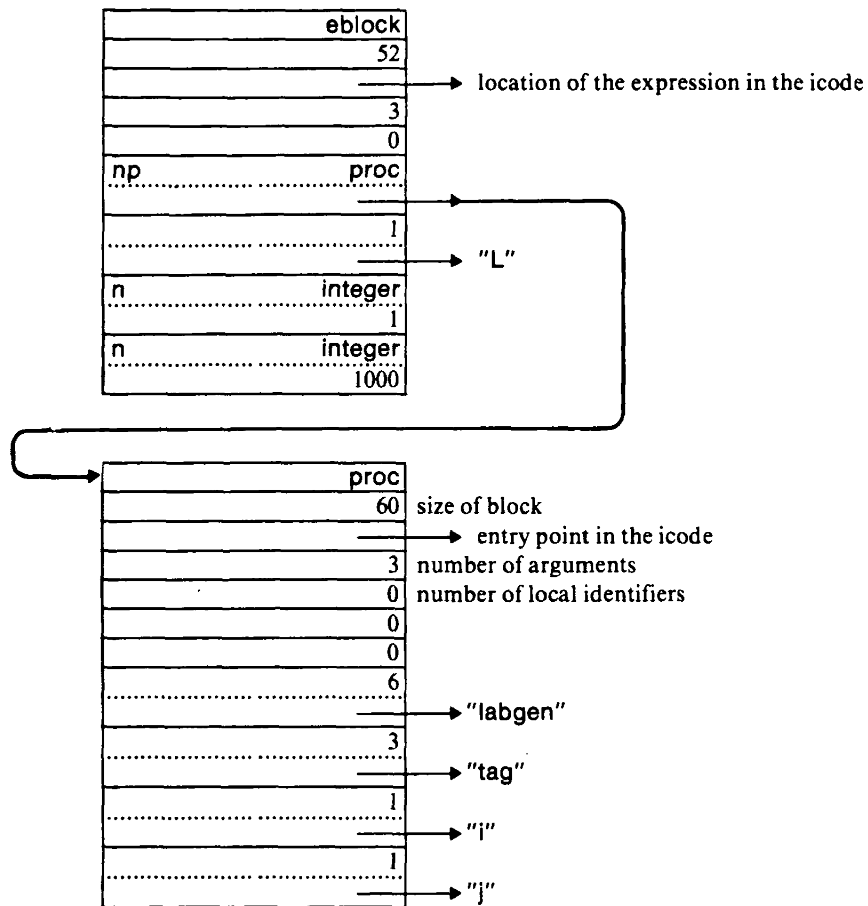
The refresh block is used to produce a refreshed copy of the co-expression, as in

```
e := ^e
```

which restores the values of the local identifiers in e to the values they had when e was originally created.

The refresh block contains a pointer to the location in the icode where the expression corresponding to the co-expression is located, followed by the number of arguments, the number of local identifiers, a descriptor that points to the procedure block for the procedure in which the co-expression was created, and the values of the arguments and local identifiers at the time of creation:

The refresh block and corresponding procedure blocks are:

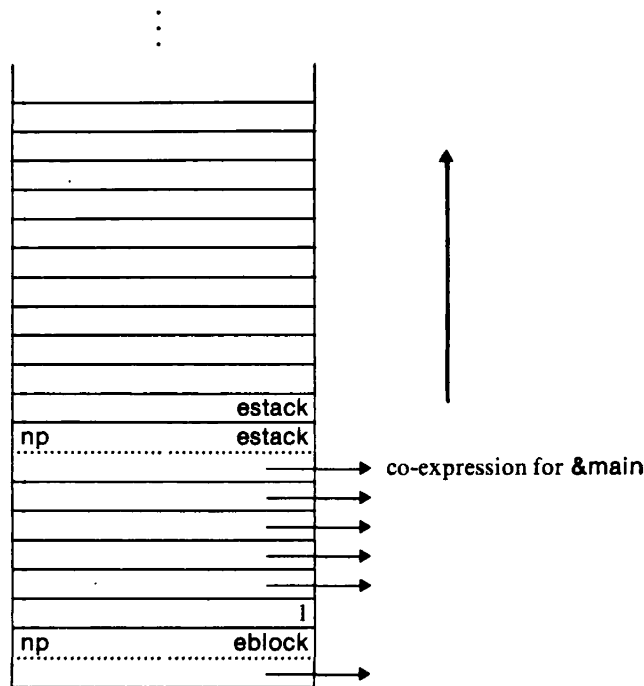


A Co-Expression Refresh Block and its Procedure Block

When a co-expression is activated, its pointer to its most recent activator is filled in by the co-expression that activated it. When it suspends, the number of results is increased by 1. For example, if

```
lab := @e
```

is evaluated in the procedure main, the co-expression for e becomes



Result of a Co-Expression Activation

Acknowledgements

Dave Hanson, Tim Korb, and Walt Hansen designed the implementation of the original version of Icon, on which many of the current structures are based [6, 7].

Cary Coutant and Steve Wampler designed the implementation of Version 5 of Icon. Most of the current data structures were done by them. Bill Mitchell has done most of the recent work on this implementation. Rob McConeghy implemented sets and revised the earlier implementation of tables in collaboration with the author.

Dave Hanson, Rob McConeghy, Bill Mitchell, and Janalee O'Bagy provided a number of useful suggestions on the presentation of the material in this report.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. R. E. Griswold, R. K. McConeghy and W. H. Mitchell, *Extensions to Version 5 of the Icon Programming Language*, The Univ. of Arizona Tech. Rep. Tech. Rep. 84-10, Aug. 1984.
3. R. E. Griswold, R. K. McConeghy and W. H. Mitchell, *A Tour Through the C Implementation of Icon; Version 5.9*, The Univ. of Arizona Tech. Rep. 84-11, Aug. 1984.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
5. W. H. Mitchell, *Porting the UNIX Implementation of Icon*, The Univ. of Arizona Tech. Rep. 83-10d, 1983.
6. R. E. Griswold and D. R. Hanson, *Reference Manual for the Icon Programming Language; Version 2*, The Univ. of Arizona Tech. Rep. 79-1a, 1980.
7. D. R. Hanson and W. J. Hansen, *Icon Implementation Notes*, The Univ. of Arizona Tech. Rep. 79-12a, 1980.