D. ASSOCIATIVE STRING ACCESS

One of the more powerful and unusual features of Snobol4 [EMM87], Icon [GRI83], and Awk [AHO78], is the ability to subscript an array with a string-valued "index," or, in general, to address data with a string-valued identifier. This section describes how this feature is implemented in patternForth.

1. Design

a) requirements

We need to be able to access data by a string identifier. This is really two problems: first, locate a string in memory given only its contents; second, somehow associate the desired data with this string, in such a way that if the string is located in memory, the data is located (directly or indirectly) as well.

Since this is intended for real-time processing application, it must be fast. The processing burden of "content addressibility" must be kept to a minimum. Methods based on sequential searches, such as the Forth dictionary, are too slow.

The string "values" of these identifiers may be quite large; several orders larger than the address size of the processor. This rules out methods based on using the string text directly as an address.

And, since this is to run on a microcomputer, the memory requirements of the content addressing scheme must be modest. The "overhead" added for fast access to strings should be minimized. In the ideal case, only the strings themselves (and their associated data) would need to be stored.

b) design decisions

The paramount requirement of speed, and the need to handle long strings, indicate a hashed storage method. The variable-length string text can be hashed into a short code, and this code used to index into a pointer table.

We would like to be able to perform the hashing function only once for each string, and afterwards use an abbreviated identifier (such as a pointer of some sort) to access the string. This confers two advantages. First, the hashing function need be performed only when the string is first identified, and not on every access to the string or its associated data. Second, if the identifying text (perhaps from some input device) is occupying temporary buffer space, this space can be freed for other use once the string has been found.

Another issue pertains to the programming model for strings. The Forth language provides virtually no support for strings of any kind, and, so there is no standard form for strings as Forth parameters. Some representation must be devised, which fits in with Forth's parameter passing mechanism (the stack), and which resembles "traditional" Forth usage.

Although an untyped language, Forth does support abstract and user-defined data types. In general, such data structures are represented and manipulated by address. This address occupies a single cell on the Forth parameter stack, allowing the use of the Forth stack operators to "rearrange" data structures in the same fashion as integer data on the stack. We would like to represent strings in a similar manner, with a single cell "address" of some kind on the stack.

Since the requirements of dynamic memory -- particularly, compaction -- preclude the direct use of the string's address, we must devise some "synthetic" identifier which occupies a single cell.

Finally, we require that an arbitrary data item be associated with a string, in such a manner that it can be located once the string is found. An obvious solution is to store this "associated data" physically with the string. Since the common Forth practice is to represent all data types with a single cell, it would seem sufficient to reserve a single cell, adjacent to the string, for associated data. But since some Forth data types, such as long integers, require two cells, we will reserve a double cell for this "associated data."

2. The hash table

a) storage method

There are two kinds of methods used for hash tables: methods using some form of linked list for the table entries, and "open addressing" methods which place entries into a sparsely- populated table [SED83].

The "open addressing" methods are better suited to situations where the total number of elements can be estimated. Access time degrades if the population is too large a fraction of the table size; the memory consumption is wasteful if the population is too small a fraction.

The "linked list" methods are more dynamic. They are suited to a demand-based allocation of the hash table memory. The table can be expanded as long as there is memory left to be allocated, making a knowledge of the eventual table size much less important. If the table shrinks, the memory occupied by the deleted entries can be released for other uses. patternForth uses a linked list hash table.

Such a linked list is illustrated in Figure D-1. There are N distinct linked lists, one for each hash code. An 8-bit hash code implies 256 linked lists in the hash table. The headers for the linked lists are stored in a table, with the hash code as the index into that table. Given the hash code, the head of its linked list can be found directly, and the linked list can be traversed.

This "root" table is static and fixed in size; only the elements of the linked lists are dynamically allocated. But, unlike open addressing, this "preallocated" table is quite small, and is not a function of the total number of elements to be stored. (We will later find it desirable to dynamically allocate this root table as well, to allow the creation and deletion of supplementary hash tables.)

b) hash code

The hash code is critical to the performance of this storage system. Unfortunately, while many references discuss the technique of hash coding [WAI73,SED83,WIR86,LEW88], there are few "hard" recommendations for hashing functions. For general applications, where the nature of the data is not known in advance, the only hashing scheme which has been suggested is "modulo N." This computation is expensive in processor time, particularly on microcomputers whose multiply and divide functions are slow (or nonexistent).

An alternative hash code is an exclusive-or of the bytes in the string. Although it has been noted [WIR86,LEW88] that this function produces a poorer distribution of hash codes, leading to an increased frequency of collision, we have chosen it because of its increased speed of computation. This may offset the decreased speed of search caused by collisions.

(With purely random binary data, such as we expect from real-time and process-control applications, either function should produce a random distribution. The "unsuitability" of an exclusive-or hashing function pertains no doubt to normal text data, which is not randomly distributed.)

We modify the exclusive-or with a bit rotation to cuase transposed characters to produce different hash codes.

We have left provision for the modulo-N hashing function, if desired later.

The current patternForth implementation uses a 6-bit hash code. Thus there are 64 entries in the "root" table, and 64 linked lists. Given the current limit of about 4096 strings, and assuming an even distribution, we can expect up to 64 elements in each list.

c) the ascension value

Waite [WAI73] has suggested an improvement for the handling of collisions in a linked-list hash table. The hashing function is extended to produce a larger code -- in our case, 22 bits. The "extra" 16 bits of hash code are stored in the list element as its "ascension value." This yields two benefits.

First, this provides a test for colliding strings which is much faster than a text comparison. A 16 bit comparison, one instruction on the 8086 processor, very quickly eliminates most of the colliding elements in a hash list. In the current envrionment, the number of 22-bit codes vastly exceeds the number of strings which can be stored, so that if the hashing function is suitably chosen, the ascension code will eliminate all the colliding strings. (It is still necessary to perform at least one text comparison, since two different strings can produce equal hash codes.)

In effect, we gain some of the advantage of a 22-bit hash code, without needing a hash table of 4 billion elements.

The second benefit of the ascension value is more subtle. It is possible at no cost to sort each linked list of the hash table in ascending order of this value. (Hence the name.) Then, the search through the list for a particular ascension value terminates when an equal or larger ascension value is found. This cuts the length of list searches, on average, in half.

To insert a new hash element, we must search for the appropriate point in the list. There is no time penalty, since such a search must be made any time a string is added to storage.

d) hash space

From where are these hash list elements dynamically allocated?

The memory allocator described previously coudl be used to obtain space for hash list elements. But these elements, being of a fixed, known size, are amenable to a much simpler, faster, and more efficient memory allocation scheme.

A separate region of memory known as "hash space" is reserved for the hash table. This space is divided into a number of segments, each of which is the size of a hash list element. All of the "free" segments are linked into a list. Space is allocated for a hash list element by simply removing one item from this free list, and an element is "released" and returned to free space by simply adding it to the free list. There is never any fragmentation or unusable space.

In patternForth, each hash list element is 16 bytes, and this is the basic unit of hash space allocation. The current implementation reserves 64K bytes for "hash space," allowing a maximum of 4096 hash table entries (and thus 4096 strings).

e) the hash table as string index

Since there are never any unusable fragments, there is never any need to "compact" the hash space. Once allocated, all of the hash list elements are "static" -- they are never required to move to a new address in memory.

This is exactly what is necessary for an single-cell string identifier! Strings cannot be represented by their address in string space, since the address of a string may change unpredictably as a result of compaction. But the address of the hash list element, which is uniquely associated with that string, never changes as long as the string remains in existence. So, this hash element address represents a "safe," if indirect, pointer to the string, which can be held on the Forth stack or in Forth variables without danger of being invalidated.

If all accesses to strings are made via indirect references through a hash list element, then the hash table itself is the "master pointer table" for a memory compactor [PET89]. The memory compactor need only change this one pointer to a string, when that string is relocated in string space. The string itself contains a "backward link" to its master pointer, to expedite the memory compaction.

Figure D-2 shows the interrelation of these data structures in the three memory spaces, including "safe," invariant, pointers in Forth variables and on the stack.

Thus, the hash table provides the necessary support for the future addition of memory compaction to patternForth.

f) the string descriptor

The address of its hash list entry is the "handle" by which a string is referenced. PatternForth expands the function of this list element somewhat, making it into a "string descriptor."

At least three items of information reside in the hash list element: the link to the next element in the list, the ascension value, and a pointer to the string text in string space. Other fixed-size data can be stored in this element, saving one level of indirection on each access.

The length of the string is a known, single- cell (16-bit) value which is stored with every string. There is an added benefit from storing it in the hash table: it can be used as another quick "elimination test" to resolve collisions. The chances of two different strings having the same hash code, and ascension value, and length, are very small indeed.

The "associated data" for the string is also kept in the hash table, and is thus located whenever the string is found. Two cells (4 bytes) are reserved for this data.

Figure D-2 shows the contents of a string descriptor. There are two cells which have not been described: one of these is a hash table pointer, to be discussed later; the other is an "access count," reserved for future work with garbage collection.

g) the hash table "root"

The "root" of the hash table is the table which contains the headers of the 64 linked lists. This table is stored in the same memory space as the list elements. (In the 8086, this limits the number of cross-segment references which are necessary.) We could have reserved a flat table for this "root." Instead, we construct the "root" as a set of 16-byte elements, allocated from the hash space. New hash tables can be dynamically created and destroyed.

The form is a two-level tree, shown in Figure D-3. Each hash table "root" requires nine 16-byte elements, each containing pointers to eight other elements in hash space. One of the nine is the true root of this tree; it points to the other eight elements of the root. These eight "branches" contain 64 pointers -- the headers of the 64 linked lists.

Three bits of the hash code index into the tree root to select a branch, and three more bits index into the "branch" to select one of its eight pointers. Should the expansion of the system require a "broader" hash table, a third level can be added to this tree, for a total of 512 hash lists.

3. Multiple hash tables

SNOBOL4 allows the definition of TABLEs, and Awk defines "arrays," which use string-valued subscripts. Two such tables are distinct, and store different values at the same string "index." This ability is frequently necessary, so it is mandatory for patternForth.

a) implementation decisions

One way to implement a TABLE is to use a single context-addressible memory space, and make the table name a prefix to the index string. For example, the construct CAPITOL["ILLINOIS"] could be translated into a associative reference to "CAPITOL^ILLINOIS", which would be distinct from "POPULATION^ILLINOIS."

There are many problems with this approach. First, it is very difficult to identify all of the members of the table. For example, to delete the CAPITOL table, the string space must be purged of all strings of the form "CAPITOL^...", requiring an exhaustive search. Second, the replication of this text prefix vastly increases the amount of storage needed. Third, this approach cannot gain speed by restricting the associative search to a known subset of the total string storage.

Instead, we create a distinct hash table for each TABLE, and add operators which specify a particular hash table for string searches.

b) defining a new hash table

The dynamic allocation of a hash table "root" has already been mentioned. Nine elements are allocated from the hash space, and the 64 list headers are initialized to "empty." The hash table search procedures are unchanged, except that we now specify which "root" is to be used. Any number of separate linked lists may coexist in the hash space.

Note that the true root of this nine-element tree, which uniquely locates the hash table, is an 16-byte element in the hash space. Except for its content, it looks just like a string descriptor, and, like a string descriptor, it can be safely referenced by its address in hash space -- a single-cell value. Note also that a hash table and a string descriptor can never have identical "handles" (addresses).

c) specifying the hash table: the string context

To specify which hash table to use for string searches, we store the "handle" of the hash table in a Forth variable, $CONTEXT ("string context").

We wish to give names to hash tables. We could use named Forth variables for this, but, to remain consistent, we refer to each hash table by association with a string.

Since it is common that the same string "name" is used for both a data value and a table, we do not use the "associated data" field to hold the "handle" of the hash table. A separate "associated hash table" field is added to the string descriptor. Thus we could have a hash table named "CAPITOL", and some integer or string data at the "address" CAPITOL as well.

d) the default hash table

There is an unnamed, "default" hash table. This table holds those strings which are not within a named hash table -- including the names of user-defined tables. Since this hash table is unnamed -- and names, anyway, change their context when a new hash table is selected -- we need an operator which selects this default table.

The first user-defined hash tables are "members" of the default hash table. Additional hash tables can be defined within these "subordinate" tables, producing a tree of hash tables, as shown in Figure D-4.

e) resemblance to the Forth vocabulary structure

The Forth "vocabulary" groups a set of related Forth words in the Forth dictionary. It defines the "context" for each dictionary search. Multiple vocabularies allow a single name to be used for several different Forth words.

The "current" vocabulary for dictionary searches is specified by the variable CONTEXT.

When the vocabularies are "sealed," a given word must be found in the current vocabulary, or the search fails. There is usually an "escape" mechanism to return to the default, "root" vocabulary.

In all of these aspects, the tree-structured vocabulary system used in many Forths parallels the patternForth TABLE.

f) the table operators

The syntax for table operations is a mixture of common Forth usage and SNOBOL4. A table is created, initialized, and associated with a string identifier with the phrase

```
" identifier" TABLE
```

It is not necessary to use a string literal; anything returning a string descriptor will be accepted by TABLE.

A table is made "current" for string searches by the { operator.

```
 " identifier" {
```

After this operator, all string searches will use the hash table associated with "identifier."

The default table is restored with the } operator. This causes the sequence

```
" tablename" { " string" }
```

to locate "string" in the table named "tablename", and then restores the default table for further searches.

Tables contained within tables are accessed as

```
" maintable" { " subtable" { " string" } }
```

Strictly speaking, only a single } is required to restore the default hash table; we use two purely for appearance.

4. Creating strings

We now describe the procedure by which strings are originally created, stored in string space, and indexed into the hash table.

a) the algorithm

To create a new string:

1. check to see if string already exists. If it does, there's no need to create it.

In our content-addressible storage scheme, a string is identified solely by its text. There cannot be two different strings with the same text; a particular text identifies a unique storage location.

2. The string needs to be created. Allocate memory for it in string space.

3. If no string space can be allocated, abort with an error flag.

Future implementations may attempt to compact the string space at this point.

4. Allocate an element from hash space for the string descriptor.

5. If no hash space can be allocated, release the allocated string space and abort with an error flag.

6. Set up the pointers to and from the string descriptor, and the other string information.

7. The search of step 1 calculated the hash code, and located its position in the ascension-ordered linked list. Insert the new element into the list.

There is no need for a separate "find string" operator, since this is performed as the first step of "create string." The two are merged into one operator, whose function is:

If the string exists, return its string descriptor. Otherwise, create it, and return its descriptor. If it doesn't exist and can't be created, abort.

b) the operators

The "get string" operators accept a string text and return a string descriptor. The text can be supplied in two ways: as a "string literal" embedded in the program, or as "raw" string data contained in a buffer.

The " operator builds string literals. It accepts text up to a closing quote mark, as:

```
" string text"
```

It is "state-smart," which means it can be either executed from the command line or compiled into a Forth word. When executed, it will find or create the string and return its descriptor. When compiled into a word, the string text is compiled as an in-line literal in the Forth code; no attempt is made to find or create the correpsonding string until the Forth word executes.

The text in a string literal cannot include an ASCII " mark; also, many Forth systems will filter out ASCII control characters. This form is not useful for "strings" of binary data.

Strings whose text is not known at compile time can be found/created with the string operator >$ ("to-string"). This operator expects an address and length of string data on the stack; it returns a string descriptor on the stack. >$ can construct strings having any arbitrary content.

c) comparison to Forth mass storage

This string access mechanism parallels Forth's BLOCK operator for access to mass storage.

BLOCK expects a physical disk block number on the stack. It returns the address of a memory buffer which contains that block, as follows:

If the block has already been read into memory, return the address of the buffer where it resides. If not, make a buffer available, read the block into that buffer, and return the buffer address.

There is never an abort due to "buffer not available" because the Forth block storage system can always empty a buffer for re-use.

5. Destroying strings

The string access mechanism "hides" the details of string search and creation from the programmer: he requests a string, and it is there.

If storage were infinite, it would never be necessary to remove a string. In real systems, though, it will become necessary to reclaim the space for other uses.

a) the algorithm

The deletion algorithm assumes that the string to be deleted is identified by content; i.e., a string search must be performed.

1. check to see if the string exists. If it doesn't, there's no need to delete it.

2. we have the string descriptor and the element which links to it (a byproduct of the search). Relink the hash list to remove the descriptor.

3. release the string space used by this string.

4. return the string descriptor to the free element pool.

b) the operator

The operator to delete a string is \$ ("wipe-string"). It expects the address and length of the string text, in the same manner as >$. This is the form most likely to be used, to delete "variable" string data which was input and processed by the program.

We presume that string literals will rarely need to be deleted, since they are generally part of the permanent structure of the program. Should it be necessary to delete a literal string, either the string operators can be employed to obtain a copy of its text in a buffer, or a new "literal delete" operator can be written.

6. The associated data field

An "associated data" storage cell is physically associated with, and logically identified by, every string.

a) the operators

Once the string descriptor is known, following a string find/create operation, the data which is associated with the string is accessed with the following operators:

`$@` given a string descriptor, returns the contents of its associated data field. ("string-fetch")

`$!` given a data value and a string descriptor, stores the data as the string's associated data. ("string-store")

These operators have been defined so as to "hide" the actual location of the associated data field. Future versions may move this data field from its current location in the string descriptor, to the allocated string space, or even to a third memory space.

b) data which can be stored

Any data which is a single cell, such as one item on the Forth stack or the contents of a Forth variable, can be stored in the associated data field. For example:

integers - a count or other numeric value may be associated with a string. One example of this [EMM87] is a count of all of the occurences of each word in a file.

addresses - addresses in Forth systems are usually single cell values, and most Forth data structures are identified by address. Thus, an arbitrary data structure can be associated with a string.

disk blocks - likewise, disk block numbers are usually a single cell, and could be associated with a string. (This has powerful database potential.)

string descriptors - string descriptors are single cell values. This allows one string to be associated with another!

And, since the "stored" string has its own associated data field, a "chain" of associative references can be built. To cite another interesting example [EMM87], states and state capitols can be associated with each other:

```
" Springfield" " Illinois" $!

" Illinois" " Springfield" $!
```

Actually, two cells are reserved for associated data. At present, double-cell operators have not been needed in patternForth, and so have not been added to the lexicon.

7. Garbage collection

Many systems which use dynamically allocated memory for the storage of working data [GRI86, GRI81,TOW86,ALL78,HEN80] perform automatic garbage collection.

a) purpose of garbage collection

Dynamic storage systems tend to fill during the course of program execution, even though the program itself, at any stage of its processing, is using only a finite and predictable amount of data. This is because of "dead" data -- data which is occupying space in storage, but is not being used.

The purpose of garbage collection is to identify this dead data, and to delete it and recover the space that it consumes.

Garbage collection should be fast, particularly in the real-time applications such as ours.

b) when can a string be deleted?

Two questions must be answered in order to collect garbage:

* When is a stored data item "dead"; i.e., when can it be safely deleted?

* How can the "dead" items be quickly found?

The first question might be rephrased, "when can stored data be deleted with no loss of information?" This is not the same as asking when the data is no longer in use. Consider the string access methodology described in this paper. We may think of the "information content" of a stored string as having two parts: the string text, and its "associations." These associations include any data (including tables) which is associated with the string, and any references to this string from another data object, i.e., any pointers to this string.

If there are no associations, the only information that this string offers is the string text. But this is precisely what must be known to gain access to the string in the first place! So, there is no information gained by accessing the string, and, in effect, the stored string offers no information at all.

Or, to view it another way: recall that the string reference operator will create a new string if it is not found. This new string will have no associated data, and will have a new and unique string descriptor. So, if there is no data associated with the string anyway, and nothing else is referring to it by its string descriptor, there is no function lost by deleting the string, and re-creating it when it is used again!

A string which references nothing, and which is not itself referenced anywhere, can be safely deleted.

Contrariwise, if any data object anywhere is referring to this string by its descriptor, the string can not be deleted, since that would invalidate the string descriptor (by returning the hash list element to the free pool), with potentially catastrophic results the next time a reference is attempted via that descriptor. And, of course, deletion of a string will cause the loss of its associated data.

c) explicit garbage collection

The simplest solution to the problem of garbage collection, is to have the programmer identify when a particular item is safe for deletion.

This offers several advantages:

* the "overhead" of garbage collection is minimal, and can be controlled and well distributed;

* we take advantage of the programmer's knowledge of the data and the application. The programmer is presumed to know the "best" time to delete the data, neither too soon nor too late.

A mark and sweep collector may wait longer than necessary to recover a "dead" string; a reference counter may recover it too soon. A programmer might choose to "preserve" a string whose references have been deleted, but which will be reused shortly, avoiding the overhead of re-creating the string.

The first advantage is of especial importance in a real-time application, where the ability to "schedule" delays can be invaluable. Most applications have "background time," known to the programmer, where such time-consuming actions as garbage collection can be performed without detriment. Explicit garbage collection allows the programmer to take advantage of these periods.

The disadvantages, of course, are:

* the burden that is placed on the programmer; and

* the chance that the programmer may fail to delete "dead" data; which, in the extreme case, could cause the memory to fill and become unusable.

Allen [ALL78] reports that explicit garbage collection in the LISP environment is "disastrous," largely due to the complex and interconnected references which are made to list elements. This problem should not exist with our simpler data structures.

The current implementation of patternForth employs explicit garbage collection.

d) mark & sweep

"Mark and sweep" is a commonly used method of garbage collection [ALL78,HEN80,GRI81]. It is named for its two phases:

* the "marking phase" in which all the data items in memory are examined, and all which are currently active are marked; and

* the "sweep phase" in which the unmarked items are deleted and their space recovered (conceivably, compaction could be performed at this point).

This "identification by omission" strategy derives from the LISP storage management system, in which explicit references can be found to all of the cells which are in use, but there is no explicit identification of those which are not in use.

The fact that this strategy must exhaustively tag every active data item in the system, in order to find those which are not active, precludes its use on a frequent basis. This garbage collection is usually invoked when a memory request fails, and more space is needed.

This is the major disadvantage of mark and sweep: it causes no overhead most of the time, but, when invoked, it brings all activity to a halt while the storage space is reorganized. This period be several seconds long, and will occur at an unpredictable time. A real-time system cannot tolerate such a suspension of processing, and certainly not on a random basis!

e) reference counting

Feucht [FEU89] describes an alternative, incremental garbage collection strategy. This strategy employs a "reference count" in every allocated data item. Whenever a reference is made to this item, the count is incremented; whenever a reference to this item is deleted, the count is decremented. When the count is zero, the item may safely be deleted.

By checking an item's reference count after every decrement, it can be deleted and its space recovered individually. This incremental approach never needs to examine all of memory at one time, and so it avoids the lengthy disruptions introduced by mark and sweep.

Feucht describes the disadvantages:

> The drawbacks to reference counting are that it is less efficient overall and causes the program to run slower due to management overhead....The mark and sweep technique is attractive with a large memory, since some programs may never invoke garbage collection and yet run faster. [FEU89]

We anticipate the real-time considerations to take precedence in patternForth, and we are using relatively small memory spaces. So, we have left provision for the future addition of this technique.

f) the problem of reference tracking

Before any automatic garbage collection can be added to patternForth, one problem must be solved: that of keeping track of all of the references to a particular item.

This problem arises because of the loose and untyped nature of Forth's "normal" storage of data and pointers. No distinction is made between an address and an integer in Forth. No mechanism exists for keeping track of all variables or other defined data objects, or for identifying the type of data stored in them. Since string "handles" have deliberately been designed to be manipulable as any other Forth data, string references cannot be adequately identified or tracked.

Two examples illustrate the problem.

First, consider a string descriptor put on the stack. Once it is on the stack, it can be DUPed, shuffled, and DROPped until any number of copies may exist anywhere within the stack. Since DUP and DROP don't know string handles from integers, there is no way to cause them to record a change in the reference count. And a marking algorithm, charged with the problem of finding the string references on the stack, could not distinguish them from integers or other addresses.

Second, consider Forth variables. Once a string handle is stored in a variable, it can be fetched any number of times; the fetch operator can't identify a string handle and doesn't know to increment a reference count. Even worse, a store to this variable will destroy the reference, but the store operator can't know this. And, since no "master list" of variables -- or worse, abstract data types -- exists, a marking algorithm would have no way to search for string references in Forth variables.

One possible solution would be to define a special set of operators -- DUP, DROP, store, and fetch -- specifically for string descriptors. This puts a burden on the application programmer, to always use the stack or memory operator which is appropriate to the data type in use, but it will provide the information needed for reference counting.

Another possible solution would be a separate stack, a special "string variables," for string descriptors. Again, a special set of operators would be necessary to manipulate the string descriptors. This could maintain the "bookkeeping" information needed by a mark and sweep garbage collector.

8. Future work

a) improved hash code

An improved hash function, perhaps using the modulo N function, can be easily added to the string package.

If this is done, it will be important to determine the impact on performance: the added overhead of hash calculation, the improvement in the hash distribution, and the speed improvement obtained through this better distribution. These statistics, of course, are dependent upon the actual data being used. So, work on improved hash codes will be postponed until some "real" applications are in use.

b) garbage collection

Automatic garbage collection will surely be desired eventually.

At present, the reference counting technique seems to be better suited to patternForth applications, so provision has been made for this technique. It will be necessary to define new operators, differentiating string processing from "ordinary" Forth processing. Exactly which operators are needed, and what restrictions need to be imposed on the programmer, remain to be determined.