Strings, Associative Access, and Memory Allocation

N. Solntseff
Department of Computer Science and Systems
McMaster University
Hamilton, Ontario, Canada L8S 4K1

Bradford Rodriguez
T-Recursive Technology
55 McCaul Street, #14
Toronto, Ontario, Canada M5T 2W7

Abstract

The goal of the present paper is the implementation of strings modelled after SNOBOL4 **[GRI71]**. Extensions to Forth to implement the memory-management strategy needed to allow strings to grow and shrink in an arbitrary manner are described. Associative or content-addressable storage needed to allow indefinitely many levels of indirectness is discussed. We propose the use of hash and string spaces, separate from the ordinary Forth space, to implement associative memory in a relatively simple manner and to allow strings to freely migrate in string space. Lastly, string store and fetch operations are introduced.

Introduction

The lack of a widely accepted model for string handling in Forth has led to a variety of attempts to implement a data structure that corresponds to the notion of *string* defined as a sequence of characters. Although strings have been included in most languages from the earliest days of computing, numerically oriented languages, such as APL, C, FORTRAN, and Pascal, have tended to relegate strings to second-class status by restricting them to fixed-length arrays. This simplification is necessary to avoid the need for dynamic memory management **[MAC83]**. Early commercial programming languages, such as Fact and COBOL, provide strings as directly manipulable data objects, although their primitive operations are also restricted to simple assignment and editing operations **[NIC75]**. In symbol-processing languages, such as LISP and Prolog, strings are regarded as list structures and do not appear as a separate data type **[MAC83]**. LISP is noteworthy in two respects: It introduced the notion of complete equivalence between program and data and the concept of dynamic storage management, including garbage collection **[PRA75]**.

Another group of languages that must be mentioned are the string-processing languages consisting of the SNOBOL family **[NIC75] [PRA75]** and its successor ICON **[GRI83]**. Pattern matching is the primary operation in SNOBOL4 **[GRI71]** in which a string is scanned to establish the presence or absence of a substring specified by a pattern expression. If the scan is successful, i.e., the substring sought is found within the string, various actions can be defined to take place, including the replacement of the matched string, assignment to variables, and function calls. Pattern matching is an extended implementation of Markov algorithms **[MAR54]** which form one aspect of the theory of computability. SNOBOL4 shares with LISP the ability to translate and execute programs that are either constructed with string-processing operations or read as string data.

In SNOBOL4, a hash table is used to store all strings, including identifier names, statement labels, or simply data. As character strings are created during execution they are entered into this table if they do not already exist. Arrays, patterns, executable code strings and program statements, as well as programmer-defined data structures are stored in a heap storage area. Mark and sweep garbage collection and full compaction is used for heap management [GRI74].

Previous Work

A string in Forth is a sequence of up to 255 bytes preceded by a byte count whose address is commonly the reference to the string value. Thus, primitive Forth string operations involve only fixed-length strings. There have been several attempts to make use of Forth's ability to define new defining words to construct a string-manipulation vocabulary. For example, Winfield [WIN83] defines a byte-string variable which contains a field for the maximum length the string can attain, as well as the actual length at any time. Operations on strings defined by Winfield include fetch, store, display, input, and output.

Other approaches revolve around the implementation of a string stack. For example, McCourt and Marisa [MCC81] have a string stack separate from the parameter stack for storing byte sequences by means of the operation

```
" ◇ <text string>"
```

where ◇ represents at least one space-character. A complete set of string-stack operations paralleling the standard data-stack set are implemented. A similar approach is made in [KEN86] which introduces a string stack and 51 words for string and string-stack operations. Finally, [BRA88] is a third example of a string-manipulation package, this time an implementation of a MUMPS-like environment with a string vocabulary of 75 words.

Other reports on string extensions can be found in the literature — there are over 13 citations in *A Bibliography of Forth References* [MAR86] dealing with some aspect of strings. In all of these, string storage is either fixed, or is implemented as a last-in-first-out string stack.

The Present Work

The goal of the present work is to implement a vocabulary of Forth extension words that can be used to specify pattern matching in the manner described by Griswold et al. [GRI71], i.e., by means of pattern-valued expressions constructed from pattern literals, pattern-valued variables, pattern-returning function calls, and the pattern producing operations of alternation and concatenation. There are two implications of this requirement: The first is that string values can grow and shrink in an arbitrary manner during the pattern matching process. The kind of memory management needed for this is far beyond that currently available in most Forth systems (for example, see [TIN86] for a detailed discussion and thorough analysis of one public-domain Forth system). The second implication is the need for associative or content-addressable storage, where string-valued data can be referred to *by the string itself* and not via an address of a memory cell or location.

Dynamic Memory Allocation

In order to make this work as widely available as possible, we use Laxen and Perry's F83 [LAX85] whenever details of a specific implementation are needed. It should also be noted that any machine dependencies refer to the Intel 8086/8088/80286 microprocessor and PC-compatible systems running under MS-DOS. (MS-DOS is a registered trademark of Microsoft

Corporation, Redmond, WA 98073, U.S.A.) Memory allocation is available in the MS-DOS operating-system environment via software interrupts. These functions (see Table 9.1 of **[DUN88]**) are called by user programs, the command processor, and by MS-DOS itself to achieve dynamic allocation, resizing, and release of memory blocks. The memory area into which programs are loaded for execution is called the Transient Program Area (or TPA). The TPA is organized into a structure called the *memory arena* which is divided into chunks called *arena entries* which can be as small as one *paragraph* (16 bytes) or as large as the entire TPA. One of the system calls is used to specify the memory allocation strategy that is to be used; the default is first fit, where the memory block at the lowest address large enough to satisfy the request is allocated. If the free block selected is larger in size than needed, then it is split up and the unused portion is returned to the free-memory pool. This leads to memory fragmentation and the possibility that memory requests may not be honoured for lack of sufficient contiguous free memory (see Chapter 5 of **[PET85]**).

We can overcome memory fragmentation by using a compacting memory management scheme in which small free blocks are periodically merged into larger ones. Symbolic processing systems incorporating Lisp translators periodically mark used memory blocks, so that the unused ones can be returned to the free-memory pool. Efficient garbage collection algorithms are known **[WAI73]**, but we felt that, initially, a simpler approach would be sufficient for a prototype system. As we are interested in memory management to provide a flexible string-storage mechanism, we can ignore most of the complexity of a general compaction algorithm such as, for example, that described in **[PET89]**. A suitable algorithm, found in **[KNU68]**, uses the first-fit method for allocation, but combines a block being returned to the free pool with an adjacent free block if one exists. It is called the *boundary tag method* by Knuth and is proposed for Forth memory management in **[SCH89]**.

In this method, every memory block, be it free or allocated, has a *tag* at both ends (see Fig. 1a). The tag field contains a numeric value representing the length of the memory block in bytes, and a Boolean flag specifying whether the block is currently in use (not free). When a memory block is no longer being used, the tags of both adjacent blocks can be examined and, if they are free, they can be joined together by adjusting the two tags at each end of the newly formed free memory block (see Fig. 1b).

Implementation of the Boundary Tag Method

As our prototype implementation is for an Intel 8086/8088/80286 microprocessor, the term *string space* refers to a 64-Kbyte memory segment which is separate from the 64-Kbyte segment occupied by Forth in the F83 system **[TIN86]**. The smallest memory block that can be allocated is two bytes in length when a 16-bit tag field is used (in this case, a single tag represents both ends of a block). This restriction of the memory-allocation unit to two bytes also ensures that single bytes are not left free during allocation. Moreover, since all lengths are even, the least significant bit can now be used as the Boolean flag.

A flow chart of the allocation algorithm is shown in Fig. 2. Here, as suggested by Knuth **[KNU68]**, the search for the first free memory area large enough to satisfy the current request is started from an address specified by a *roving pointer*, here called **rover**, which always points to the tag byte immediately after the last allocated block. At present, memory deallocation is not performed automatically when the system runs out of memory, but is performed as the result of an invocation of the procedure **release**. Its flow chart is shown in Fig. 3. It should be noted that there is no provision for compaction in the memory allocation and release described here, although, to be perfectly general, access to memory blocks should allow for a possible future inclusion of compaction. This is described in the next section.

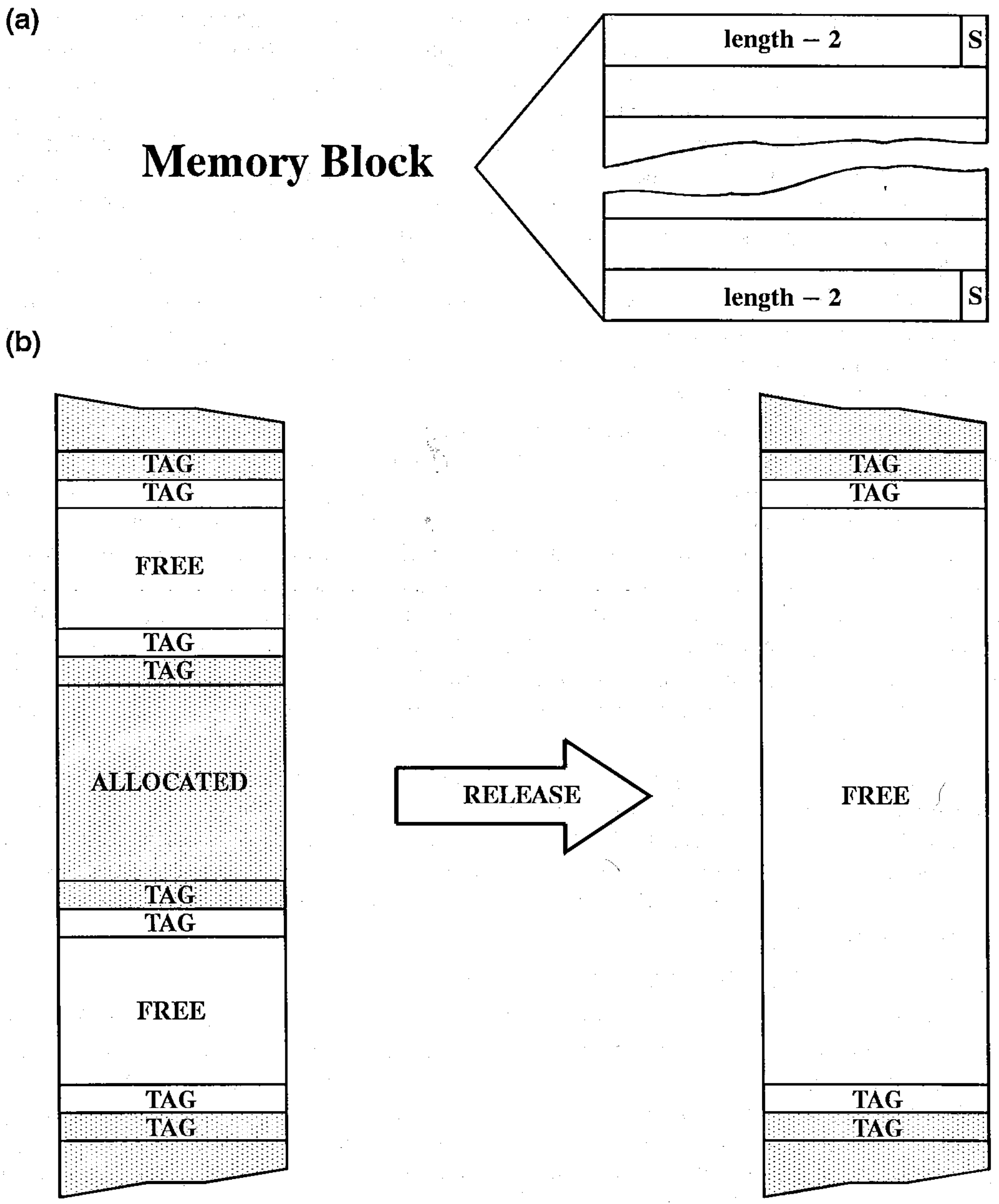

Figure 1. The Boundary Tag Method of Memory Management. (a) The structure of a single memory block. *Note:* The length of the block is always even and less than 65536. The Status bit S is zero if the block is free; otherwise it is one. (b) Amalgamation of adjacent free memory blocks.

Access to Strings

As the goal of the present work is to introduce strings into Forth as closely as possible to the SNOBOL4 model, it is natural to manage the 64-Kbyte string space as a heap with individual strings addressed via a hash table. Hashing schemes fall into two broad categories: in the first, entries are stored in a large array, whereas in the second, they are linked together by means of pointers **[SED83]**. In the present project, we adopted the latter approach because of its greater flexibility. The actual hashing algorithm we chose is described in Chapter 6 of **[WAI73]**.

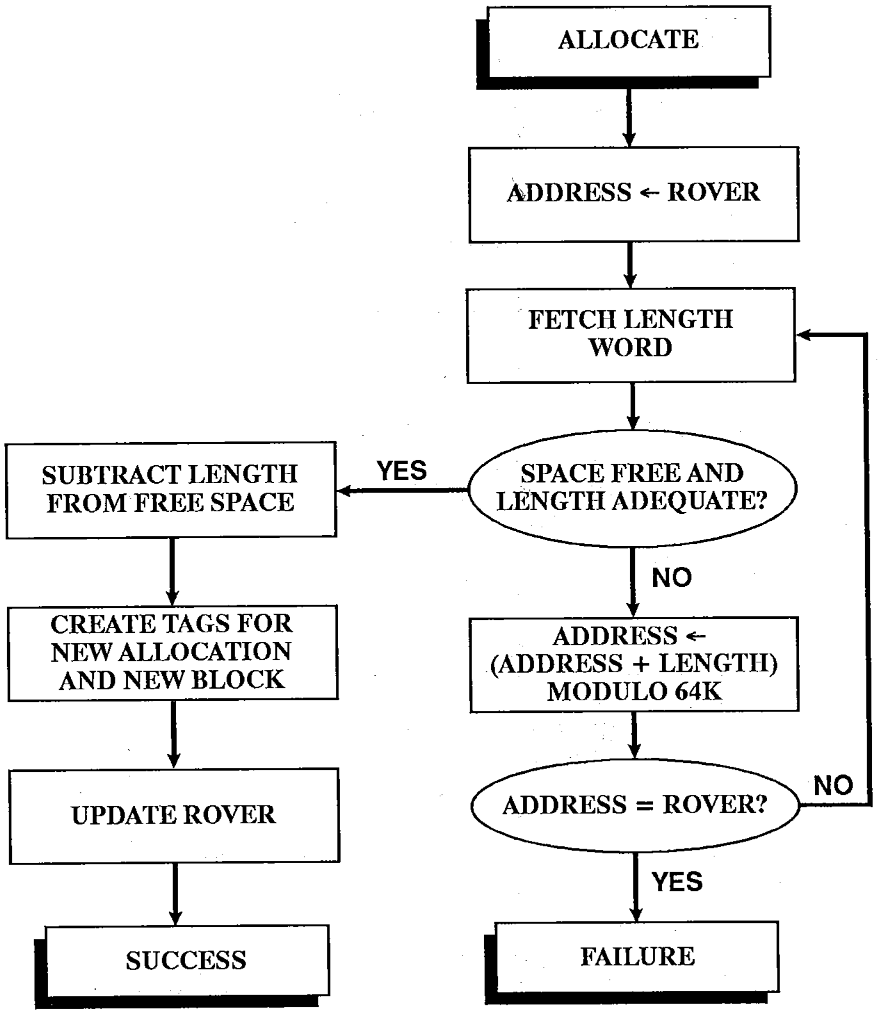
ALLOCATE
ADDRESS ← ROVER
FETCH LENGTH WORD
SPACE FREE AND LENGTH ADEQUATE?
YES
SUBTRACT LENGTH FROM FREE SPACE
CREATE TAGS FOR NEW ALLOCATION AND NEW BLOCK
UPDATE ROVER
SUCCESS
NO
ADDRESS ← (ADDRESS + LENGTH) MODULO 64K
ADDRESS = ROVER?
NO
YES
FAILURE

Figure 2. Flow Chart of the Memory Allocation Algorithm.

A hash code is generated from a string and used as an index into a hash table with N chains, where each chain is a linked list of descriptors containing pointers to the string-space heap (see Fig. 4). For each string, Waite's algorithm obtains a supplementary hash code which he calls the *ascension value*. The ascension value provides a first test to resolve collisions arising from the initial hashing and only if this fails does it become necessary to resort to string comparisons. Entries in each collison chain are stored in ascending order of the ascension value (hence its name) and this strategy halves the average search time. The insertion procedure is only slightly lenghthened by the need to determine if the new string's ascension value is already present in the collision list.

Our primary hash index is 8 bits long and provides an index into a table of 256 collision chains. The ascension value is 16 bits long to allow for a single-instruction compare on the Intel 80x86 microprocessors, so that our effective hash code is 24 bits long. All entries in a collision list are of the same size (see Fig. 5), so that they can be efficiently allocated from and returned to a separate collision-list storage pool. It should be noted that no garbage collection is necessary. The hash table and the collision lists are stored in a separate 64 Kbyte segment which is called the *hash space* (see Fig. 6).

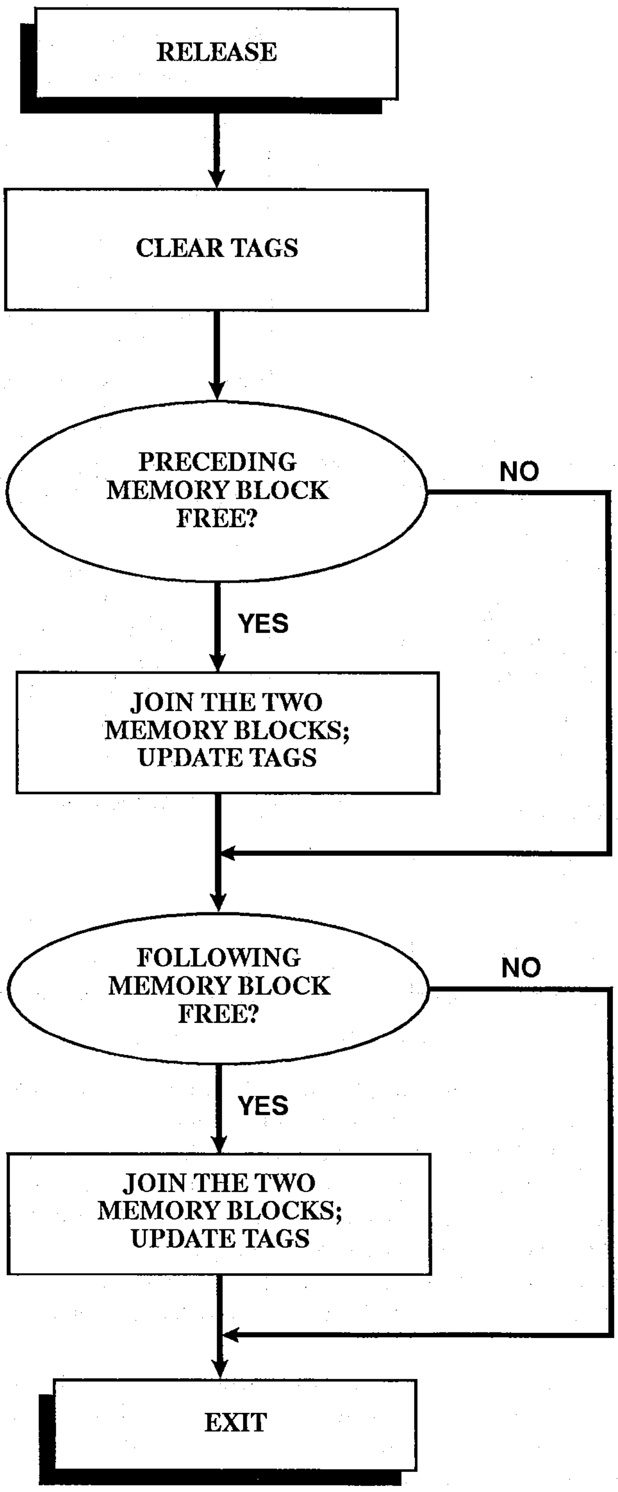
RELEASE
CLEAR TAGS
PRECEDING MEMORY BLOCK FREE?
NO
YES
JOIN THE TWO MEMORY BLOCKS; UPDATE TAGS
FOLLOWING MEMORY BLOCK FREE?
NO
YES
JOIN THE TWO MEMORY BLOCKS; UPDATE TAGS
EXIT

Figure 3. Flow Chart of the Memory Release Algorithm.

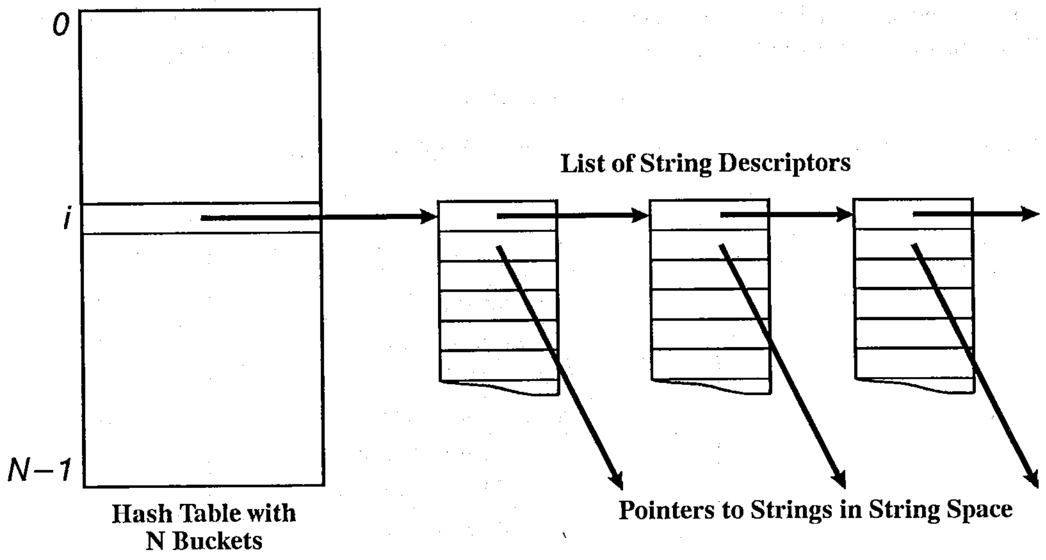
0
i
N−1
List of String Descriptors
Hash Table with N Buckets
Pointers to Strings in String Space

Figure 4. Hash Table Implemented with Linked Lists.

Operations on Strings

Any extension of Forth to include strings should allow string values to be treated in the same manner as any other Forth entity, be it an address, integer, or other value. In the case of strings as implemented in the present paper, this requirement is met by associating a unique *handle* with each string. The handle is a 16-bit value that can be stored, fetched, and manipulated as any other single-precision Forth quantity. It is convenient to take the address of the string descriptor in hash space as the handle. It should be noted that the handle remains unchanged when the string value it identifies moves to a new location in string space on account of compaction.

Strings are both created and referenced by means of the **$"** operator as follows:

```
$" <text string>"    ( – handle)
```

where the text string is read from the terminal input buffer and the handle is left on the data stack. If the string already exists in string space, its handle is returned; otherwise, the string value is created in string space and a new string handle is issued. Compatibility with the string type found in most Forths is achieved by the operator **>$** (called *to string*):

```
>$ a n    ( – handle)
```

where **>$** expects an address and length of a text string on the data stack and returns an existing or a newly issued handle. Other string functions that must be included in the final package are those dealing with operations such as concatenation, substring extraction, pattern matching, and so on. These are described elsewhere **[ROD89]**.

Strings in our model, as in SNOBOL4, can be used as identifiers, that is, as names for variables capable of storing arbitrary data, including other strings. This potential for indefinitely many levels of *indirectness* is one of the reasons for the continued use of SNOBOL4. The fourth field of string descriptor in our implementation (see Fig. 5) is a 16-bit value which may be an address, an integer, or another string handle. The last possibility allows any string to be the value of another string. The value associated with a string can be accessed by means of the operator **$@** (or *string fetch*)

```
$@    ( handle – n)
```

where **handle** is a string handle and **n**, the value of the string handle when used as an identifier. The converse operation of assigning **n** to string **handle** is performed by a *string store*

```
$!    ( n handle -)
```

which places **n** into the appropriate field of the descriptor of string **handle**.

BYTE OFFSET	15 0
0	pointer to next in collision chain
2	ascension value
4	pointer to string
6	data associated with string
8	reserved for future use
10	reserved for future use
12	reserved for future use
14	reserved for future use

Figure 5. A String Descriptor or an Element in a Hash Table Collision List. *Note:* The fourth field of the string contains the data associated with the string when it is used as an identifier.

Summary and Conclusions

The goal of the present work is the implementation of strings modelled after SNOBOL4 strings and the introduction of pattern matching into Forth by means of pattern-valued expressions constructed from pattern literals, pattern-valued variables, pattern-returning functions, and the pattern operations of alternation and concatenation.

Extensions to Forth to implement the memory-management strategy needed to allow strings to grow and shrink in an arbitrary manner are complete, although compaction remains to be included. Associative or content-addressable storage is needed to allow indefinitely many levels of indirectness when strings are treated as identifiers. The use of hash and string spaces, separate from the ordinary Forth space as shown in Fig. 6, implements associative memory in a relatively simple manner and allows strings to freely migrate in string space. This aspect of the work described in this paper is also complete.

A comprehensive set of string operations such as concatenation, substring extraction, and testing, as well as the inclusion of patterns, pattern expressions, and primitive pattern functions as provided in SNOBOL4 remain to be implemented.

References

[BRA88] R. Braithwaite, "Using a Forth stack," *Forth Dimensions,* X, No. 3 (September/October 1988), pp. 15-37; *Forth Dimensions,* X, No. 4 (November/December 1988), pp. 30-36.

[DUN88] R. Duncan (General Editor), *The MS-DOS Encyclopedia,* Microsoft Press, Redmond, WA 98073 (1988), 1570 pp.

[GRI71] R. E. Griswold, J. F. Poage, and I. P. Polansky, *The SNOBOL4 Programming Language,* (Second Edition), Prentice-Hall, Englewood Cliffs, N.J. 07632 (1971), 256 pp.

[GRI74] R. Griswold, *The Macro Implementation of SNOBOL4,* W. H. Freeman, San Francisco, CA (1972).

[GRI83] R. E. Griswold and M. T. Griswold, *The Icon Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632 (1983), 313 pp.

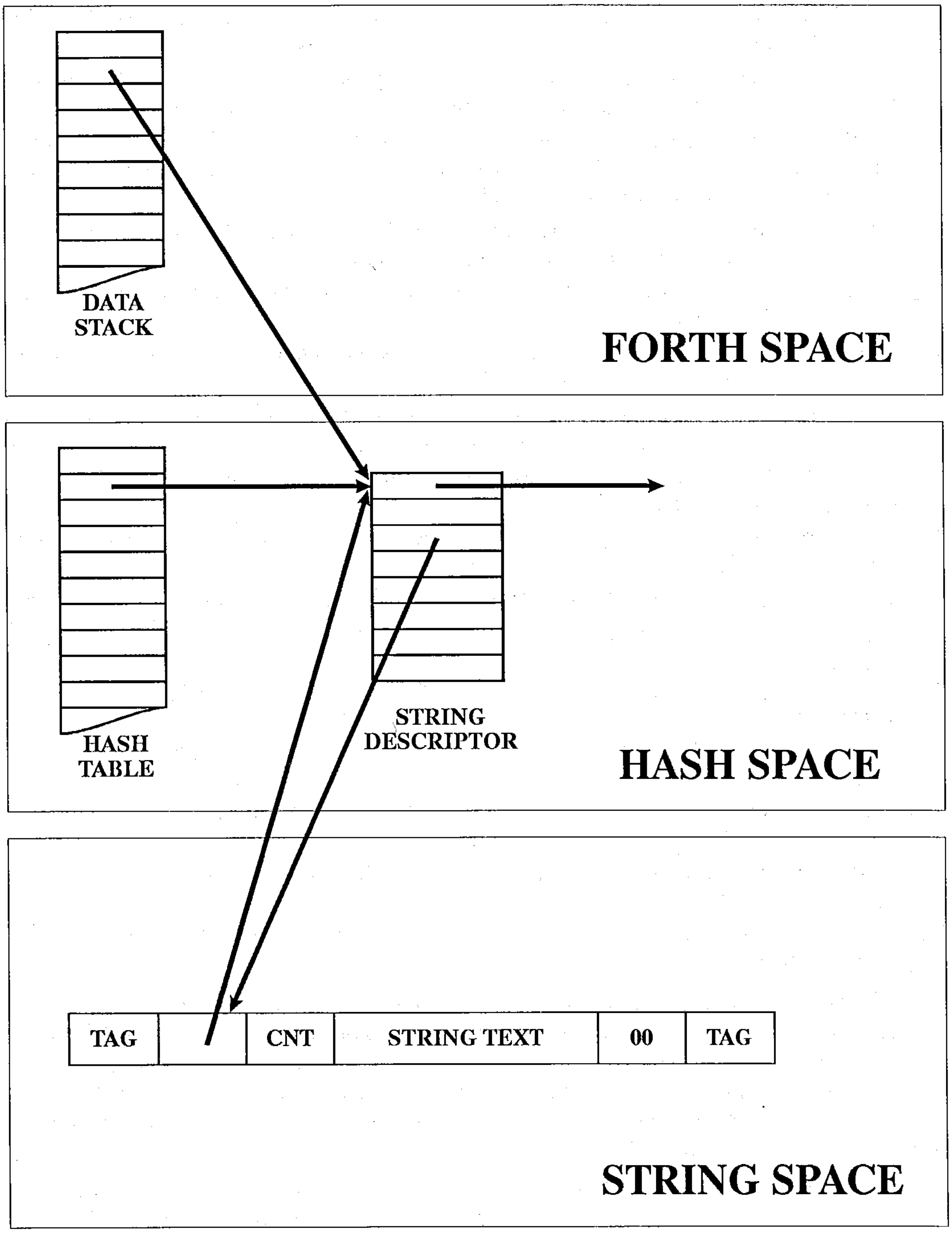

Figure 6. The Relationship Between Forth Space, Hash Space, and String Space.

[KEN86] C. Kent, "F83 string functions," *Forth Dimensions,* 7, No. 6 (March/April 1986), pp. 23-33.

[KNU68] D. E. Knuth, *The Art Of Computer Programming, Volume 1, Fundamental Algorithms,* Addison-Wesley Publishing Company, Reading, Massachusetts, (1968), 634 pp.

[LAX85] H. Laxen and M. Perry, *F83 Source,* Offete Enterprises, Inc. San Mateo, CA 94402 (1985), 209 pp.

[MAC83] B. J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation,* Holt, Rinehart and Winston, New York, N.Y. 10017 (1983), 544 pp.

[MAR54] A. A. Markov, *Teoriya Algorifmov,* Akademiya Nauk SSSR, Moscow (1954); translated as: *Theory of Algorithms,* U.S. Department of Commerce, Office of Technical Services, Document No. OTS 60-51085 (1961), 444 pp.

[MAR86] T. Martin (Editor), *A Bibliography of Forth References,* Third Edition, Institute for Applied Forth Research, Rochester, NY 14611 (1986), 167 pp.

[MCC81] M. McCourt and R. A. Marisa, "The string stack," *Forth Dimensions*, III, No. 4 (November/December 1981), pp 121-124.

[NIC75] J. E. Nichols, *The Structure and Design of Programming Languages,* Addison-Wesley Publishing Company, Reading, MA (1975), 572 pp.

[PET85] J. L. Petersen and A. Silberschatz, *Operating System Concepts,* Addison-Wesley Publishing Company, Reading, MA, Second Edition, (1985), 625 pp.

[PET89] S. Peterson, "A memory allocation compaction system," *Dr. Dobb's Journal,* **14**, No 3 (April 1989), pp. 50-56 and 82-90.

[PRA75] T. W. Pratt, *Programming Languages: Design and Implementation,* Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632 (1975), 530 pp.

[ROD89] B. Rodriguez and N. Solntseff, "A new implementation of strings in Forth," Technical Report, Dept. of C.Sc. and Systems, McMaster University, Hamilton, ON L8S4k1 (1990), in preparation.

[SED83] R. Sedgewick, *Algorithms,* Addison-Wesley Publishing Company, Reading, MA (1983), 551 pp.

[SCH89] K. Schleisiek, "Dynamic Memory Allocation," *1988 FORML Conference Proceedings,* Forth Interest Group, San Jose, CA 95125 (1989), (to be published).

[TIN86] C. H. Ting, *Inside F83* (Revised Edition), Offete Enterprises, Inc. San Mateo, CA 94402 (1986), 287 pp.

[WAI73] W. M. Waite, *Implementing Software for Non-Numeric Applications,* Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632 (1973), 510 pp.

[WIN83] A. J. Winfield, *The Complete Forth,* Sigma Technical Press, Wilmslow, Cheshire, U.K. (1983), 130 pp.

Dr. Nicholas Solntseff was born in Shanghai, China of Russian emigrant background. He received his PhD in high energy Physics from the University of Sydney, Australia in 1958 and has worked in nuclear engineering in England. He has been teaching computer science since 1970 and has held University appointments in England, Australia, USA, and Canada. His current interests are semantics of programming languages and medical applications of expert systems, especially for preoperative risk assessment.

Brad Rodriguez received an M.S. in Electrical Engineering and an M.S. in Computer Science from Bradley University. Currently he is pursuing a PhD in Electrical Engineering at McMaster University, focusing on real-time control applications of AI technology, and consulting part-time as T-Recursive Technology, which specializes in embedded microprocessor systems. Current Forth projects include a multiprocessor Super8 lighting control system, and a pattern-matching extension of Forth. He is also writing a book on metacompilation and embedded programming in Forth, and is promoting a new Forth Implementation Team to create public-domain Forths for new microprocessors.