An Alternate Forth Dictionary Structure

James C. Brakefield
Technology Incorporated, Life Sciences Division
300 Breesport, San Antonio, Texas 78216

Abstract:

The data structures used for definitions and the word search of Forth can facilitate various utilizations of the same. My goal is completeness and efficiency.

Constants or literals are stored separately. This is so they can be hashed. The concept of a constant is enlarged so that the code string is a constant in the same sense as a text string. A constant is something which is self-identifying and unchanging at the textual level.

A colon definition is then a pairing of a code string with a name. Only pointers to the two "constants" are kept in the definition entry (a pointer to the name string and a pointer to the code string).

The various constant and definition records are implemented with tagged fields. This allows inspection of the entire record by any routine possessing a pointer into any field of the record.

Additional optional fields are: Reference count, Comment pointer, Input type list pointer, Result type list pointer, etc.

Talk:

I have been rethinking the memory structure of Forth in order to achieve certain goals. This effort is similar to the work of several other people, RTL being the closest in spirit. The driving constraint is that Forth definitions always be decompilable. At the same time, provision is made for various "advanced" features (multiple cfa's, type checking, precedence parsing, overloading).

The problem of decompiling a code string with embedded constants becomes continually more difficult if the user is allowed to add new forms of constants. Thus it seems that the embedded constant is something of a black sheep to be discouraged in the same sense that occurred with the GOTO. As one looks at code strings one sees that the relative branch displacements are also embedded constants. This has a major effect on Forth control structures but fortunately there is a nice solution.

As one examines the various ways word definitions can be arranged in memory, one eventually gives up as there is no fixed record structure which gives all the features one wants. Thus, I eventually settled for a less memory-efficient but expandable record structure. By the use of tags each field is self-defining. Thus a symbol table entry can be "understood" given a pointer to any of its fields.

The issues of whether tags should be addresses or just codes, whether tags are even needed, this is left unresolved. At issue is how to recognize a field and what that recognition consists of. Coded tags

are limited to a small set of values. If one wants to add another type of tag, one may need to add another bit to the tag data structure. This can have no end. Using a full address for the tag solves this problem but at a memory cost which may be unacceptable. Letting the field identify itself through indirection, while in the true spirit of Forth, may require constraints which make difficult the flexibility and variability desired.
The subject of treating code as a constant has generally been recognized, but not utilized. The dichotomy of program and data has so far been too great to unify by the ordinary programmer. (Ask a programmer how he likes coding his control constants). Although both Forth and Lisp allow one to manipulate program as data, only Prolog actually makes program and data look the same. This is probably due to Prolog's lack of sequentuality, i.e., one tends to think of programming as coding the sequential activity of the CPU whereas data has an inherent timeless nature.

(My technique for representing code strings is to use the curly brackets to surround a code string. This lets code strings nest, but as the bracketed code string is replaced by its address, this causes no problem. As each right bracket is encountered the code string becomes complete and is saved in free space as an unnamed code string constant. You can think of the brackets as one letter BEGIN and END. This makes Forth code strings look like Lisp or Prolog data lists.)

What all of this leads up to is that the 32-bit microprocessor is at hand and the 64-bit microprocessor looms in the not-so-distant future. We all know that Forth is the best way to use the beast. However, there is no agreement on how to organize memory. Files should not be necessary as the very few systems will have anywhere near four gigabytes of disk. Thus, screens can be attained some other way, preferably by decompilation. By treating code strings as constants, they can be hashed and reused without being named. One can have a situation where code is continually being written and added to the system. Those pieces of unnamed code which have unsuspectedly high reference counts become candidates for naming.

Appendix A:

Dictionary Entry Field Tags:

Primary code field (parameter and code fields)
Link field -- for hashed or linear search
Secondary code field -- for multiple cfa's
Length of code & parameter field -- for storage reclamation
Reference count field
Name pointer field (name elsewhere)
Code field pointer (definition elsewhere)
Parameter type list pointer
Result type list pointer
Comment string pointer
Immediate flag (accomplished via two versions of the link field tag)
Raisability flag (accomplished via two versions of code field tags)
Precedence & parsing field -- for algebraic syntax

Last code field & extension pointer (if non-zero) -- allows continuation

Appendix B:

Definitions:

Code string: List of word addresses preceded by DOCOLON and followed by EXIT. I.e., a headerless definition.

Constant: A data structure (i.e., contents of memory) which does not change once created.

Hashing: Technique for speeding a search for a given item. Some function of the item is computed which tends to yield a flat histogram of all items. This is used to subdivide the items and hence restrict the search.

Immediate: A word which is executed by the compiler rather than being compiled.

Literal: A non-address constant embedded within a code string. Considered by the author as in same category as the GOTO.

Raisable: A word which may be executed without indirection. Non-raisable words must precede their parameter fields and thus require at least one level of indirection from a code string which references them.

Type: A word, which given the address of a datum of its type, computes the length of that datum.

Word: Named or unnamed executable code string. If named, its textual form is the text form of the name string; if unnamed, its textual form is as a bracketed code string.

Appendix C:

Tokens:

(Another area of concern is that of the naming of Forth words. Forth's use of the white space as the only name separator is certainly general. I tend to prefer the more typical algebraic tokens wherein the character set is divided into categories and characters in each category form tokens which do not always need a white space for separation. This usually leads to the categories of numeric constants, alphanumerics, non-alphanumerics, and singletons.)

Decimal integer:	digit string -- 1234
Octal integer:	"octal digit string -- "773
Hexidecimal integer:	$hexidecimal digit string -- $FF01
Boolean integer:	%binary digit string -- %11011
Real:	integer with embedded decimal point -- 1.5
Character string:	'character string' -- 'this is a string'
Alphanumeric:	alphabetic letter, alphanumeric string -- AM2905
Non-alphanumeric:	non-alphanumeric string -- <>

```
Singleton:              single character token (separators & brackets)
Code string:            token or {word address list} -- { SWAP DROP }
Comment:                ~ text carriage return -- ~ gibberish
Token binder:           _
(Alphanumerics, non-alphanumerics, and singletons may be combined with
the underscore:  A5b_,_++ is a token)
```

Appendix D:

Programming Language Hierarchy:

Non-extensible machine language or micro-code
  Non-stack oriented programming
  Many programmers operate at this level even if computer supports stacks

Extensible machine language -- Forth
  Two stacks, micro-codes higher level facilities

Traditional Algebraic -- Fortran, Pascal, C, ADA
  Both data structure and control must be planned
  Usually single stack implementation
Dynamic memory allocation -- Lisp
  Data structuring automatic, control must be planned

Dynamic control -- Prolog, Expert Systems languages
  Both data structure and control automatic
  Forward versus backward chaining
  Probablistic control often used

References:

1) Buege, Bob. Status Threaded Code. 1984 Rochester Forth Conference, p103.

2) Solley, Evan. An In-circuit Development System with a Forth Heritage. 1984 Rochester Forth Conference, p25.

3) Steel, Guy L., Jr. Common Lisp. Digital Press, 1984.

4) Clocksin & Mellish. Programming in Prolog. Springer-Verlag, 1981.

5) Dijkstra, E.W. GOTO Statement considered harmful. Communications of the ACM, 11:3:147-8, March 1968.

(6) Elliott, I.B. The EPN and ESN Notations. Sigplan Notices, 19:7:9-17, July 1984.