Symbolic Stack Addressing

Adin Tevet
P. O. Box 217
44101 Kfar Sava
Israel

Data on the stack can be given symbolic names for fetching and storing while not significantly increasing execution times. Symbolic names compile addresses relative to the top of the stack, thereby avoiding the run-time cost of setting up and dismantling stack frames. Data is fetched from the stack by **PICK** and stored into the stack by a new word **POST**. The compiler can know the run-time stack height from the change in stack height induced by each word in the dictionary.

Introduction

FORTH code could be written more easily, could be read more easily, and could avoid excessive use of local variables if values on the stack were referenced symbolically for fetching and storing. As a simple example of the usefulness of symbolic stack addressing, consider how a high level definition for the word **MAX** might look:

```
: MAX ( n1 n1 --- n ) OVER OVER < IF SWAP THEN DROP ;
```

The wording of this definition has no intrinsic relationship to the meaning of "maximum." Instead it describes a sequence of stack operations that produce the desired result. For more complex definitions the sequence of operations can be difficult to follow. By contrast, consider the following definition using symbolic stack addressing:

```
: MAX IN( N1 N2 ) N1 N2 < IF N2 ELSE N1 THEN => N OUT( N ) ;
```

In this definition the expression **IN(N1 N2)** assigns the symbolic names **N1** and **N2** to the two values at the top of the stack. These values can now be referenced symbolically. The code **N1 N2 < IF N2 ELSE N1 THEN** describes the relationship between **N1**, **N2** and their maximum value in terms that are relatively easy to follow. The expression **=> N** gives the name **N** to the result of the computation and **OUT(N)** outputs this value, replacing the inputs **N1** and **N2**. For more complex definitions the improvement in readability when symbolic names are used is even more apparent.

Symbolic stack addressing can be implemented using stack frames **[1-3]**. As each word is invoked, the location on the stack of its bottommost argument on the stack is recorded as the boundary of its stack frame. The compiled code references arguments on the stack as offsets relative to this location. When the word terminates, the calling word's stack frame is restored. There is a run-time cost to this process and to other methods of implementing symbolic stack addressing. Execution times might increase by 50% **[4]**, 20–30% **[5]**, 20–30% **[6]**, or 10–30% **[7]**.

Perhaps the best eventual method for introducing symbolic stack addressing into FORTH will be a hardware implementation of stack frames. Until such machines appear, any proposed software method will fail to gain acceptance if it imposes too heavy a penalty on the execution time of words. The approach presented here is to reference stack items from the top of the stack, rather than from a stack frame location. This avoids the run-time cost of stack frames. It does however require that the stack height be known at compile time. Knowing the run-time stack

height at compile time had turned out to be easier than expected. Using this approach, execution times generally increase by less than 10% and sometimes decrease.

Method of Implementation

The following approach is used to determine the distance from the stacktop to a value referenced by a symbolic name. A "stack height counter" maintains the stack height at compile time. Each colon definition is responsible for maintaining only its own height, so that the "height" is only with respect to the word being defined. In order to maintain the correct value in the stack height counter all run-time changes to the stack height must be known. If we examine the FORTH-83 standard [8] we see that almost all FORTH words have a fixed stack effect at run time, and therefore cause a fixed change in the stack height. We define the "gain" of a word to be the change in stack height induced by executing the word. For example, the gains for the words **LOAD**, **NEGATE**, and **HERE** are (respectively) -1, 0, and +1. We will also refer loosely to the "gain" of a sequence of words. As each word is compiled into a colon definition its gain is added to the stack height counter, so that the counter shows the stack height after the word is executed.

Symbolic names are defined within a temporary dictionary during compilation of a colon definition. When a symbolic name is defined the current stack height is entered into its definition as the stack location of the name. When the symbolic name is later referenced for fetching or storing, the difference between its stack location and the current stack height is the offset from the stacktop that is compiled with **PICK** for fetching or compiled with **POST** (a new word) for storing. The compiled code will then consist of **PICK**s from the stack, **POST**s into the stack, and other FORTH words executed on the topmost elements of the stack as arguments.

Unknown Stack Heights

The apparent difficulty with this approach is that the run-time stack height could become unknown at compile time. The two parts of an **IF-ELSE-THEN** might have unequal gains, or the number of iterations of a loop might be determined by run-time data, or a word without a well-defined gain (such as **?DUP**) could be used, thereby causing the stack height to become unknown.

Because of regularities in the way that FORTH is coded an unknown stack height is much rarer than might be expected. The two parts of an **IF-ELSE-THEN** usually have equal gains, so that the stack height is known at compile time regardless of which branch will be taken at run-time. The body of a loop usually has zero gain, so that the stack height is known regardless of the number of executions of the loop body. Words with undefined gains are uncommon. **?DUP** and **-'** being the only such words listed in the FORTH-83 standard. These regularities are so consistent that it is worthwhile to receive compiler warnings of a violation as an indicator of a probable programming error. Even intentional violations tend to fall into patterns. An **IF-ELSE-THEN** whose parts have unequal gains usually contains a "jump" word (**LEAVE**, **QUIT**, etc.) in one part while the other part describes the "normal" branch. A loop with nonzero gain is usually followed by another loop with nonzero gain, and the stack height becomes known again after the second loop. A **?DUP** is usually followed by an **IF** which results in a known stack height. New words **THIS-HIGH**, **THAT-HIGH**, **]HIGH**, and **?DUP-IF** (described in the next section) handle these patterns to avoid an unknown stack height.

But suppose that the stack height does become unknown, so that the stack height counter holds an incorrect value. Since the offset compiled with **PICK** or **POST** is computed as a difference of stack heights, this error will be cancelled out for any symbolic name which is defined after the stack height became unknown. The only situation in which the offset might be computed incorrectly is if the stack height became unknown after the name was defined. This means that the stack location corresponding to the name is buried at an unknown stack depth and cannot

be accessed in any case, with or without symbolic stack addressing. Any method for accessing the data that is used without symbolic stack addressing (e.g. placing it in a variable or on the return stack) can also be used with symbolic stack addressing (see **"]HIGH"-EXAMPLE** in the next section). Thus there are no new difficulties introduced by symbolic stack addressing.

Extending FORTH for Symbolic Stack Addressing

The words that extend FORTH for symbolic stack addressing are presented here.

The word **IN(** ("in-args") is followed by a sequence of symbolic names, one for each input argument, and terminated by a right parenthesis. If there are no input arguments, then **IN(** should be followed by one or more blanks and a right parenthesis.

The word **OUT(** ("out-args") is followed by a sequence of symbolic names, one for each output argument, and terminated by a right parenthesis. If there are no output arguments, then **OUT(** should be followed by one or more blanks and a right parenthesis.

The defining word => ("define-symbol") defines a new symbolic name and assigns the current stacktop location to it. A fetch from this stack location is compiled when the symbol subsequently appears by itself in the code.

These three words are illustrated in the following definition of **QUADRATIC:**

```
: QUADRATIC-#1  ( a b c x --- a*x*x+b*x+c )
  IN( A B C X ) A X * B + X * C + => QUAD OUT( QUAD ) ;
```

There is a restriction on the use of **OUT(**. No value can be moved by **OUT(** to a higher location on the stack. For example, the following definition of **ROT** will not work correctly because **N1** is being moved to a higher location on the stack:

```
: ROT IN( N1 N2 N3 ) OUT( N2 N3 N1 ) ;
```

The following correct definition first copies **N1** (so that the stack is **N1 N2 N3 N1**) and then moves the desired output values to positions that are not higher on the stack;

```
: ROT IN( N1 N2 N3 ) N1 => N1' OUT( N2 N3 N1' ) ;
```

An easy way to comply with this restriction is to output the values in the same order as they were input or created, i.e. in the same order as they exist on the stack (skipping values not needed for output). Note that the correct definition of **ROT** follows this rule. Also note that **OUT(** used with no arguments or with one argument is always correct.

The defining word **=>(** ("define-symbols") defines a sequence of symbolic names terminated by a right parenthesis. The last of these symbols corresponds to the current stacktop.

A symbolic name used by itself compiles a fetch from the stack location referenced by the name. If however the symbol is preceded by the word **TO>** ("to-symbol"), then a store into the stack location is compiled. When the symbol is preceded by the word **TOP>** ("top-symbol"), then code is compiled to reset the stacktop to the stack location corresponding to the symbol.

These words are found in the following definition of the Greatest Common Denominator:

```
: GCD IN( A B ) A B < IF B A ELSE A B THEN =>( LARGE SMALL )
      BEGIN    LARGE SMALL /MOD =>( REM QUOTIENT )
 REM  WHILE    SMALL TO> LARGE  REM TO> SMALL  TOP> SMALL
      REPEAT   OUT( SMALL ) ;
```

TOP> can also be followed by **Ø** to reset the stacktop to the stack location just preceeding the first input argument, leaving "zero-arguments" on the stack.

The word +!> ("plus-store-symbol") preceding a symbolic name will compile code that increments the value found in the stack location corresponding to the name. The increment is taken from the stack at run time. The word TRAVERSE illustrates the use of this word:

```
: TRAVERSE  IN( ADDR DIRECTION )  \ direction = +1 or = -1
   BEGIN DIRECTION +!> ADDR  128 ADDR C@ < UNTIL  OUT( ADDR ) ;
```

When using the CREATE-DOES> construct, the gain of the CREATE-part is the gain of the defining word, while the gain of the DOES>-part is the gain of the defined words and should be stored in the gain field of each new definition. The words CREATE-GAIN and DOES>-GAIN will perform these tasks when used in the form:

```
: ... CREATE ... CREATE-GAIN DOES> ... DOES>-GAIN ;
```

This is illustrated in the following definition:

```
: ARRAY  CREATE  IN( #ROWS #COLS )
        #ROWS ,  #ROWS #COLS * ALLOT  OUT( )
      CREATE-GAIN
      DOES>  IN( ROW COL PFA )  PFA @ => #ROWS
       #ROWS COL * ROW + PFA + 2+ => ADR  OUT( ADR )
      DOES>-GAIN ;
```

The gain of a newly-defined word is determined by the compiler whenever possible, but there are certain situations in which the gain must be explicitly set. The next three definitions illustrate how GAIN can be used to set the gain of a deferred word, a code word, and a recursively-defined word:

```
DEFER KEY  1 GAIN
CODE DROP  AX POP  NEXT  END-CODE   -1 GAIN
: FACTORIAL  RECURSIVE  [ Ø GAIN ]  IN( N )  N 1 =
   IF 1 ELSE N 1- FACTORIAL  N * THEN => N!  OUT( N! ) ;
```

if symbolic stack addressing is being used or is GAIN-WARNINGS is TRUE then the compiler issues warning in situations that can cause the stack height to become unknown. The remaining four words are concerned with these situations. They are seldom actually needed.

The words THIS-HIGH and THAT-HIGH are used in IF-ELSE-THEN expressions whenever it is necessary to designate which part of the expression to use for determining the stack height. The IF-part and the ELSE-part usually produce the same stack height and neither of these words is needed. (if no warning is issued, then neither word is needed). If one of these words is needed, then it should be placed before THEN. THIS-HIGH selects the height of the part just before THEN. THAT-HIGH selects the height of the other part. For example,

```
IF QUIT ELSE .S THEN
```

does not require either word, while in each of the following expressions the height is taken from the part not containing QUIT:

```
IF QUIT    ELSE .         THIS-HIGH THEN
IF .       ELSE QUIT      THAT-HIGH THEN
IF . QUIT                 THAT-HIGH THEN
```

The word]HIGH ("BRACKET-HIGH") resets the stack height counter after an unknown stack height becomes known again. Suppose, for example, that we wish to accept N characters from the keyboard, then emit them, and finally print the number N, where N is at the top of the stack and is not known until run time. In this example the stack height becomes unknown after the first loop and becomes known again after the second loop.

```
: "]HIGH"-EXAMPLE        \ Stack height known at compile time:
  IN( N )                \ height = 1
  N >R                   \ height = 1 +1 -1 = 1
  N Ø DO KEY LOOP        \ height = 1 +1 +1 -2 +N = 1+N = ?
  R> Ø DO EMIT LOOP      \ height = 1+N +1 +1 -2 -N = 1
  [ 1 ]HIGH              \ Reset the stack height counter to 1
  N . OUT( ) ;
```

The word **?DUP-IF** ("question-dup-if") avoids a situation in which the stack height becomes unknown because the gain of **?DUP** is undefined. It can be used in place of the sequence **?DUP IF** (two words).

Stack Awareness

Colon definitions using symbolic stack addressing can be written with very little awareness of the stack. One can simply assign symbolic names to the input arguments of the colon definition and to any intermediate values, reference arguments by name as needed, and finally output the desired results by name. With greater stack awareness, some words can be coded for increased run time efficiency without sacrificing readability, as in the following definition of **QUADRATIC**.

```
: QUADRATIC-#2  IN( A B C X )
  A X * B + ( x a*x+b ) * ( c a*x*x+b*x ) + => QUAD
  OUT( QUAD ) ;
```

Timing

The readability of a definition may be increased by using symbolic stack addressing, but its execution time might also be increased. For example, the definition of **QUADRATIC-#1** using symbolic stack addressing (see above) is easier to read than a "standard" definition of **QUADRATIC** [9]:

```
: QUADRATIC ( a b c x --- N )  >R SWAP ROT R@ * + R> * + ;
```

However, **QUADRATIC-#1** compiles the execution sequence

```
3PICK OVER * 3PICK + OVER * 2PICK + (NIPS) 8
```

which requires more time for execution than the standard version.

In order to better understand the tradeoffs involved, several of the words defined above have been times against a standard published version of the same word. Both versions were compiled by Laxen and Perry's F83 [10] and any I/O in the original has been supressed for these tests. With execution times normalized to 1.00 for the standard version, we get 1.07 for **QUADRATIC-#1** (but 0.90 for **QUADRATIC-#2**), 1.09 for **GCD** [11], 0.92 for **TRAVERSE** [12], 1.29 for an array word defined by **ARRAY** [13], and 1.02 for **SIEVE** [14] (code not shown). These figures suggest that FORTH code written with a **PICK** and **POST** implementation of symbolic stack addressing is generally less than 10% slower than "standard" FORTH code. Since no extra run-time mechanisms (such as stack frames) are used, the additional time is presumably due to a larger number of words (and corresponding invocations of **NEXT**) in the definitions written with symbolic stack addressing as compared to the standard definitions. Note that **OUT(** causes no unnecessary movement of data. For example, the definition

```
: SYMBOLIC  IN( N1 N2 N3 )  OUT( N1 N2 N3 ) ;
```

executes exactly as does a standard null definition

```
: STANDARD ;
```

The definition of Perkel's **BOX [15]** uses many stack manipulation words. The execution time for a version using symbolic stack addressing (code not shown) is 0.86, suggesting that symbolic stack addressing can be expected to improve the execution times for definitions that require many arguments on the stack at the same time.

Compilation time using symbolic stack addressing is 1.06 compared to compilation without symbolic stack addressing, as determined by compilations of the 54-screen file **UTILITY.BLK** in the Laxen and Perry F83 system.

Storage

The memory requirements for this implementation are 3500 bytes. About 900 bytes are used for redefining existing FORTH words, and could be largely eliminated in a future implementation.

Conclusions

Accessing stack values as offsets from the stacktop is an effective way to implement symbolic stack addressing. The execution time for the compiled code and the storage requirements for the implementation are reasonable. No auxiliary stacks, predefined local variables, or other run-time mechanisms are used. The ordinary stack manipulation words are available for greater efficiency, but are not needed solely to get an argument into position.

I believe that the novice FORTH programmer can use symbolic stack addressing to write quite reasonable code with little stack awareness, while the experienced FORTH programmer can profitably apply symbolic stack addressing to save coding and debugging time for words that require many arguments on the stack. All users will benefit from code that is easier to read.

Acknowledgements

I would like to thank Frans C. H. Schneider of the Netherlands for introducing me to FORTH. Additional thanks are due Schneider and Johann Borenstein for critiquing an earlier draft of this paper.

References

1. Paul Bartholdi, "Another Aid for Stack Manipulation and Parameter Passing in Forth," *1982 Rochester Forth Conference on Databases and Process Control*, The Institute for Applied Forth Research, Inc., 1982.
2. Hans Nieuwenhuÿzen, "Notes on the PARAMETER Description by Paul Bartholdi," *1982 Rochester Forth Conference on Databases and Process Control*, pg. 227, The Institute for Applied Forth Research, Inc., 1982.
3. George B. Lyons, "Stack Frames and Local Variables", *Journal of FORTH Applications & Research*, Vol. 3, No. 1, 1985.
4. Rieks Joosten, "Techniques Working Group", *1982 Rochester Forth Conference on Databases and Process Control*, The Institute for Applied Forth Research, Inc., 1982.
5. Harvey Glass, "Towards a More Writeable Forth Syntax", *1983 Rochester Forth Applications Conference*, The Institute for Applied Forth Research, Inc., 1983.
6. Sidney A. Bowhill, "Fast Local Variables for Forth," *1982 FORML Conference Proceedings*, Forth Interest Group, 1982.
7. S. Korteweg and H. Nieuwenhuÿzen, "Stack Usage and Parameter Passing", *Journal of Forth Application and Research*, Vol. 2, No. 3, 1984.
8. *FORTH-83 Standard*, Institute for Applied Forth Research, Inc., P. O. Box 27686, Rochester NY 14627, USA, August 1983.

9. Leo Brodie, *Starting Forth*, Prentice-Hall, 1981, page 112.
10. No Visible Support Software, P. O. Box 1344, 2000 Center Street, Berkeley, CA. 94704, USA.
11. Robert L. Smith, "Greatest Common Divisor", *FORTH Dimensions*, Vol. 2, No. 6, p.166.
12. Mitch Derick and Linda Baker, *FORTH Encyclopedia*, Mountain View Press, Inc., P. O. Box 4656, Mountain View CA, 94040, USA, 1982, Page 283.
13. Leo Brodie, *Starting Forth*, Prentice-Hall, 1981, page 297.
14. Mitchell E. Timin, "Sieve of Eratosthenes in FORTH", *FORTH Dimensions*, Vol. 3, No. 6, page 181.
15. Marc Perkel, "The Integer Solution", *FORTH Dimensions*, Vol. 6, No. 2, page 18.
16. Donald E. Knuth, "Literate Programming", *The Computer Journal*, Vol. 27, No. 2, 1984.

Adin Tevet learned FORTH five years ago while working in the Robotic Laboratory of the Israel Institute of Technology (the "Technion"). He is now a free-lance programmer.

Implementation Code

Influenced by Knuth **[16]**, the implementation code is subdivided into numbered sections, each section containing related text and source code describing one aspect of the implementation. The word **SECTION-LOAD** loads the source code found in a section. For example, **10 SECTION-LOAD** will load the source code found in section 10. This code runs on Laxen and Perry's F83 for the IBM PC [10]. In order to perform properly the gains of all non-immediate words must be stored in their gain fields, for example as follows:

```
-1 ' LOAD !GAIN     1 ' HERE !GAIN
```

Because gains are zero by default, only nonzero gains need to be explicitly set.

Diskettes with this implementation of symbolic stack addressing were distributed at the 1988 Rochester FORTH conference. Additional diskettes will be sent by the author on request. The diskette includes code to set the gains for all words in Laxen and Perry's F83 and modifications to allow decompilation of definitions that use symbolic stack addressing.

Sec. 1. Symbolic Stack Addressing

Symbolic stack addressing is a feature that allows data on the stack to be referenced by symbolic anmes. This program extends Laxen and Perry's F83 to include a method of symbolic stack addressing in which the location of data on the stack is computed as an offset from the top of the stack rather than from a stack frame. In order to know the run-time stack height at compile time a stack height counter is maintained. This counter is incremented by the "gain" of each word as it is compiled. The gain of a word is the run-time change in stack height induced by executing the word. Stack locations are stored in a temporary dictionary.

```
DECIMAL
 2 SECTION LOAD   \ Stack height counter
 9 SECTION LOAD   \ Temporary dictionary
10 SECTION LOAD   \ Symbolic names
13 SECTION LOAD   \ IN(
14 SECTION LOAD   \ OUT(
 3 SECTION LOAD   \ Maintain the run time stack height
15 SECTION LOAD   \ Gain of new colon definitions
```

Sec. 2. Stack Height Counter

Because the stack height counter is frequently used, special words are defined for accessing it. Warnings of possible error in the stack height counter are controlled by **GAIN-WARNINGS**. The stack height counter can be reset using **HIGH** or **]HIGH**.

```
VARIABLE STACK-HEIGHT
: SH!      ( n --- ) STACK-HEIGHT !    ;
: SH+!     ( n --- ) STACK-HEIGHT +!   ;
: SH@      ( --- n ) STACK-HEIGHT @    ;
VARIABLE GAIN-WARNINGS    VARIABLE GAIN-WARNINGS-SAVE
GAIN-WARNINGS ON
VARIABLE VALID-HEIGHT
: HIGH?    ( --- f ) VALID-HEIGHT @ NOT ;
: .LAST    ( --- )   LAST @ .ID ; \ Useful for warning messages
: ]HIGH    ( n --- ) SH!  VALID-HEIGHT ON ] ;
: ]+HIGH   (N ---)   SH+! VALID-HEIGHT ON ] ;
```

Sec. 3. Maintain the Stack Height Counter

Several categories of words must be considered separately for maintaining the stack height counter.

```
4 SECTION LOAD  \Words compiled into colon definitions
6 SECTION LOAD  \Immediate words (not branching)
7 SECTION LOAD  \Branching words
8 SECTION LOAD  \Defining words (not colon)
```

Sec. 4. Words Compiled into Colon Definitions

Each word's gain is stored in its gain field (Sec. 5). The word] is modified so that it adds a word's gain to the stack height counter when the word is compiled into a colon definition.

```
5 SECTION LOAD    \ The gain field
: ]-REVISED ( --- ) \ ... to update the stack height counter
  STATE ON   BEGIN  ?STACK  DEFINED DUP
  IF    Ø> IF  EXECUTE  ELSE   DUP @GAIN SH+!   , THEN
  ELSE  DROP  NUMBER  DOUBLE?
    IF       [COMPILE] DLITERAL    2 SH+!
    ELSE DROP  [COMPILE] LITERAL     1 SH+!         THEN
  THEN  TRUE DONE? UNTIL  ;
: ]-PATCH ]-REVISED ;
' ]-PATCH ' ] 6 CMOVE
FORGET ]-PATCH
```

Sec. 5. The Gain Field

Four bits are removed from the view field to be used as the gain field. The previous view field now looks like **VGVV** where **V_VV** is the remaingin view field and **_G__** is the gain field. The gain must be in the range -8 thru +7. A file can contain only 255 screens for viewing

Because the gain field is not a whole word it should be accessed using only **@GAIN** and **!GAIN** rather than with **>GAIN**.

```
: >GAIN  ( cfa --- gfa )  >VIEW 1+ ;
: GAIN>  ( gfa --- cfa )  1- VIEW> ;
```

```
: NO@ ( addr --- n )  \ Fetch least significant nibble n
  C@ 15 ( = ØF hex ) AND ;
: NO! ( n addr --- )  \ Store n as least significant nibble
  >R 15 AND R@ C@ 24Ø ( = FØ hex ) AND OR R> C! ;
: BIG? ( gain cfa --- ) SWAP -8 8 WITHIN NOT GAIN-WARNINGS @ AND
  IF >NAME .ID ." gain too big " ELSE DROP THEN ;
: @GAIN ( cfa --- gain )
  >GAIN NO@  8 XOR 8 - ( extend sign bit of nibble ) ;
: !GAIN ( gain cfa --- )
  2DUP BIG?  >GAIN NO! ;
: @VIEW2 ( cfa@ --- scr file# ) \ Change 4Ø95 to 255 in @VIEW
  >VIEW @ DUP 255 AND  DUP Ø= ABORT" entered at terminal."
  SWAP 4Ø96 / 15 AND ;
: PATCH  @VIEW2 ;  ' PATCH  ' @VIEW  6 CMOVE  FORGET PATCH
\ Two words for debugging:
: .SH  SH@ . ; IMMEDIATE       \ Sprinkle thru colon definition
: .GAIN  ( -- )  ' @GAIN  .  ;  \ Print gain of following word
```

Sec. 6. Immediate Words (Not Branching)

Immediate words must add their own gain to the stack height counter. Most immediate words, such as `\S` and `RECURSE`, have no run-time stack effect and need not be revised for symbolic stack addressing. Others are handled elsewhere: `DOES>` in Sec. 8, `;` in Sec. 15, and the branching words in Sec. 7. This implementation of symbolic stack addressing does not handle assembly code, so if an immediate word `;CODE` or `;USES` is used, then the gain of the defined word will have to be set with `!GAIN` or `GAIN`. The possible stack effects of two immediate words, `[` and `[COMPILE]`, can be too complex to handle in any general way. If used with symbolic stack addressing, then it is necessary to make the adjustments to the stack height counter for the specific case in hand, using `]HIGH` or `]+HIGH`.

```
: LITERAL       1 SH+!  [COMPILE] LITERAL      ; IMMEDIATE
: DLITERAL      2 SH+!  [COMPILE] DLITERAL     ; IMMEDIATE
: [']           1 SH+!  [COMPILE] [']          ; IMMEDIATE
: ASCII         1 SH+!  [COMPILE] ASCII        ; IMMEDIATE
: CONTROL       1 SH+!  [COMPILE] CONTROL      ; IMMEDIATE
: ?LEAVE       -1 SH+!  [COMPILE] ?LEAVE       ; IMMEDIATE
: ABORT"       -1 SH+!  [COMPILE] ABORT"       ; IMMEDIATE
: "             2 SH+!  [COMPILE] "            ; IMMEDIATE
: CANCEL ( -- ) cancel gain of next word )   >IN @
  ' @GAIN NEGATE SH+!                        >IN !  ;
: IS  -1 SH+! CANCEL [COMPILE] IS ;  IMMEDIATE
\ Prefer [MAKE] ?  Sets gain of deferred word
\ : [MAKE]  ' ' 2DUP @GAIN SWAP !GAIN  \ as in:  [MAKE] KEY (KEY)
\  [COMPILE] LITERAL >IS [COMPILE] LITERAL COMPILE ! ; IMMEDIATE
```

Sec. 7. Branching Words

An `IF-ELSE-THEN` causes the stack height to become undefined just in case the gain of the `IF`-part differs from the gain of the `ELSE`-part. If there is no `ELSE`-part, then the stack height becomes undefined just in case the gain of the `IF`-part is not zero.

If the gain of a loop body is not zero, then the stack height becomes undefined.

Whenever branching in the code causes the stack height to become undefined a warning message is displayed and a flag is set. The flag can be cleared by resetting the stack height counter using]HIGH or]+HIGH (Sec. 2). If not cleared before completion of the colon definition, then the gain of the word is undefined and another warning is displayed (Sec. 15).

The words ?DUP-IF, THIS-HIGH, and THAT-HIGH can be used in certain situations to avoid an unknown stack height.

```
: ?HEIGHT  ( previous-SH@ -- )  SH@ = NOT
   IF VALID-HEIGHT OFF  GAIN-WARNINGS @
      IF .LAST ." has undefined height " THEN THEN ;

\ Same as ?DUP IF but stack height remain known
: ?DUP-IF COMPILE ?DUP SH@ 1- [COMPILE] IF ; IMMEDIATE
\ Warn when ?DUP is used alone (undefined gain)
: ?DUP  -1 ( impossible height ) ?HEIGHT ?DUP ;

: IF   -1 SH+!  SH@ [COMPILE] IF ;  IMMEDIATE
: ELSE >R >R  SH@ SWAP SH!  R> R>  [COMPILE] ELSE ;  IMMEDIATE
: THEN [COMPILE] THEN ?HEIGHT ;  IMMEDIATE

\ For use with LEAVE, QUIT or EXIT. Place just before THEN .
\  Take the gain of the last part
: THIS-HIGH  SH@ 2POST ;  IMMEDIATE
\  Take the gain of the other part
: THAT-HIGH  2PICK SH! ;  IMMEDIATE

: ?DO  -2 SH+!  SH@  [COMPILE] ?DO ;  IMMEDIATE
: DO   -2 SH+!  SH@  [COMPILE]  DO ;  IMMEDIATE
: LOOP   [COMPILE]  LOOP      ?HEIGHT ;  IMMEDIATE
: +LOOP  [COMPILE] +LOOP -1 SH+! ?HEIGHT ;  IMMEDIATE

: BEGIN  SH@  [COMPILE] BEGIN  ;  IMMEDIATE
: UNTIL  [COMPILE] UNTIL  -1 SH+! ?HEIGHT ;  IMMEDIATE
: WHILE  >R >R  -1 SH+! SH@ SWAP  R> R>  [COMPILE] WHILE ; IMMEDIATE
: REPEAT  [COMPILE] REPEAT  ?HEIGHT SH! ;  IMMEDIATE
: AGAIN  [COMPILE] AGAIN ?HEIGHT ;  IMMEDIATE
```

Sec. 8. Defining Words (Not Colon)

A defining word must store the gain of each new definition in its gain field. CREATE-GAIN and DOES>-GAIN are for convenience when using the CREATE-DOES> construct.

```
: GAIN  ( gain --- )  LAST @ NAME> !GAIN ;
-1 GAIN  \ Apply to itself ( Gain of GAIN is -1)

: CREATE     ( --- )          CREATE      1 GAIN ;
: VARIABLE   ( --- )          VARIABLE    1 GAIN ;
: CONSTANT   ( n --- )        CONSTANT    1 GAIN ;
: 2VARIABLE  ( --- )          2VARIABLE   1 GAIN ;
: 2CONSTANT  ( nØ n1 --- )    2CONSTANT   2 GAIN ;

: CREATE-GAIN  SH@ ( Gain of CREATE-part ) Ø SH!
     COMPILE (LIT)  HERE Ø ,  COMPILE GAIN ;  IMMEDIATE
: DOES>-GAIN  SH@ SWAP ! ( Patch gain of DOES>-part )
     SH! ( Restore gain of CREATE-part ) ;  IMMEDIATE
```

Sec. 9. Temporary Dictionary

During compilation of a colon definition, a **SYMBOLIC** vocabulary holds the definitions of symbolic names. These definitions are stored in a temporary dictionary at a distance above the normal dictionary that should be sufficient to avoid conflict with the pad.

```
VOCABULARY SYMBOLIC
VARIABLE SYMBOL-DP
VARIABLE NORMAL-DP
VARIABLE NORMAL-CURRENT
VARIABLE NORMAL-LAST
: BEGIN-SYMBOLIC  ( --- ) CURRENT @ NORMAL-CURRENT !
   LAST @ NORMAL-LAST !  HERE 256 + SYMBOL-DP !
   ALSO SYMBOLIC DEFINITIONS ;
: RESET-SYMBOLIC-VOCAB  ( --- )
   ['] SYMBOLIC >BODY #THREADS 2* ERASE ;
: END-SYMBOLIC ( --- )   NORMAL-CURRENT @ CURRENT ! PREVIOUS
   NORMAL-LAST @ LAST ! RESET-SYMBOLIC-VOCAB ;
: SYMBOL-D'Y  ( --- ) DP @ NORMAL-DP !   SYMBOL-DP @ DP ! ;
: NORMAL-D'Y  ( --- ) DP @ SYMBOL-DP !   NORMAL-DP @ DP !
   NORMAL-LAST @ LAST ! ;
```

Sec. 10. Symbolic Names

The defining words **=>** and **=>(** define new symbolic names. Subsequent use of a symbolic name causes compilation of a stack accessing word together with an offset to the data on the stack. The stack accessing words are defined in Sec. 11. Words that compile the stack accessing wors are defined in Sec. 12. Because symbolic names exist only within a single colon definition, warnings about non-uniqueness are not issued.

Since the definition of **=>(** contains loop bodies with nonzero stack gains, a definition using symbolic stack addressing is presented for comparison. Note that the stack height becomes undefined after the **BEGIN-END**, yet symbolic stack addressing can continue to be used within the colon definition. The stack height becomes known again after the **DO-LOOP**, and the stack height counter is then reset by **]HIGH**.

```
11 SECTION-LOAD  \ Stack access words
12 SECTION-LOAD  \ Compile stack access words
: SYMBOL  ( --- )
   WARNING @ WARNING OFF  CREATE SH@ ,  WARNING !
   DOES> @ FETCH ;
: => SYMBOL-D'Y SYMBOL IMMEDIATE NORMAL-D'Y ; IMMEDIATE
: ) ; IMMEDIATE
LAST @ @ FORGET ) CONSTANT NAME-)-IMMEDIATE
: =>( SYMBOL-D'Y Ø
     BEGIN 1+ SYMBOL IMMEDIATE LAST @
          SWAP OVER @ NAME-)-IMMEDIATE =
     UNTIL HIDE NIP 1- Ø
     DO NAME> >BODY I NEGATE SWAP +! LOOP
     NORMAL-D'Y ; IMMEDIATE
: LOCATION' ( --- locn ) ' >BODY @ ;
```

```
: TO>   LOCATION' STORE ;  IMMEDIATE  ( See Sec. 12)
: TOP   LOCATION' RETOP ;  IMMEDIATE
: +!>   LOCATION' ADDTO ;  IMMEDIATE
\ : =>( SYMBOL-D'Y Ø
\      BEGIN
\       IN( COUNT )
\         SYMBOL IMMEDIATE LAST @    => NFA
\         COUNT 1+                   => COUNT+1
\         NFA @ NAME-)-IMMEDIATE =   => DONE?
\       OUT( NFA COUNT+1 DONE? )
\      UNTIL HIDE ( nfa-parenthesis count+1 ) NIP 1- Ø
\      DO IN( NFA ) I NEGATE NFA NAME> >BODY +! OUT(  ) LOOP
\      [ Ø ]HIGH NORMAL-D'Y ;  IMMEDIATE
```

Sec. 11. Stack Access Words

Fetching from a location relative to the stacktop is performed by the FORTH word **PICK**. There is currently no FORTH word for storing relative to the stacktop. We define a word **POST** to perform this function. If **W** is at the top of the stack, then **N POST** will store **W** as the **N**-th stack value from the top of the stack, not counting **W** itself. **POST** is a partial inverse operation to **PICK**. If **N** is any unsigned number, then **N PICK N POST** will leave the stack unchanged.

For greater run-time speed we define a word (**PICK**) similar to **PICK** except that it gets its argument from the next location in the parameter field. This argument is pre-computed at compile time to units corresponding to stack entries within the processor, so that no further computation is needed at run time. Similarly, we define (**POST**), (**DROPS**) (for resetting the stacktop), and (**BUMP**) (for incrementing a value on the stack). To save space and improve execution time, faster versions of **PICK**, **POST**, and **DROPS** are defined for their most commonly occurring arguments.

```
\ Coded in F83 Assembler for the 8Ø85
2 CONSTANT STACK-UNIT  \ One stack unit on the 8Ø86
CODE POST   ( w n --- ) BX POP  BX SHL  AX POP  SP BX ADD
        AX Ø [BX] MOV  NEXT END-CODE
CODE (PICK)  AX LODS AX BX MOV  SP BX ADD  Ø [BX] AX MOV
          1PUSH END-CODE
CODE (POST)  AX LODS AX BX MOV  AX POP
          SP BX ADD  AX Ø [BX] MOV  NEXT END-CODE
CODE (DROPS)  AX LODS  AX SP ADD  NEXT END-CODE
CODE (BUMP)  AX LODS AX BX MOV  AX POP
          SP BX ADD  AX Ø [BX] ADD  NEXT END-CODE
CODE 2PICK  ( n2 n1 nØ --- n2 n1 nØ n2 )
          SP BX MOV   4 [BX] AX MOV  1PUSH END-CODE
CODE 3PICK  ( n3 n2 n1 nØ --- n3 n2 n1 nØ n3 )
          SP BX MOV   6 [BX] AX MOV  1PUSH END-CODE
```

```
CODE 1POST  ( n1 nØ n --- n nØ )
        SP BX MOV  AX POP  AX 4 [BX] MOV  NEXT END-CODE
CODE 2POST  ( n2 n1 nØ n --- n n1 nØ )
        SP BX MOV  AX POP  AX 6 [BX] MOV  NEXT END-CODE
CODE 3POST  ( n3 n2 n1 nØ n --- n n2 n1 nØ )
        SP BX MOV  AX POP  AX 8 [BX] MOV  NEXT END-CODE

CODE 2DROPS  4 # SP ADD  NEXT END-CODE
CODE 3DROPS  6 # SP ADD  NEXT END-CODE
CODE 4DROPS  8 # SP ADD  NEXT END-CODE
```

Sec. 12. Compile Stack Access Words

```
: UNITS,  ( n --- n )  STACK-UNIT  *  ,  ;

: COMPILE-PICK  ( n --- )
        DUP  Ø  = IF  DROP  COMPILE  DUP
   ELSE DUP  1  = IF  DROP  COMPILE  OVER
   ELSE DUP  2  = IF  DROP  COMPILE  2PICK
   ELSE DUP  3  = IF  DROP  COMPILE  3PICK
   ELSE                     COMPILE  (PICK) UNITS,
   THEN THEN THEN THEN ;

: COMPILE-POST  ( n --- )
        DUP Ø  = IF  DROP  COMPILE  NIP
   ELSE DUP 1  = IF  DROP  COMPILE  1POST
   ELSE DUP 2  = IF  DROP  COMPILE  2POST
   ELSE DUP 3  = IF  DROP  COMPILE  3POST
   ELSE                    COMPILE  (POST) UNITS,
   THEN THEN THEN THEN ;

: COMPILE-DROPS  ( n --- )
        DUP Ø = IF  DROP ( ... and compile nothing )
   ELSE DUP 1 = IF  DROP  COMPILE DROP
   ELSE DUP 2 = IF  DROP  COMPILE 2DROPS
   ELSE DUP 3 = IF  DROP  COMPILE 3DROPS
   ELSE DUP 4 = IF  DROP  COMPILE 4DROPS
   ELSE                   COMPILE (DROPS) UNITS,
   THEN THEN THEN THEN THEN ;

: OFFSET-FROM-TOP  ( locn --- offset )  SH@ SWAP - ;

: FETCH ( locn ---)  OFFSET-FROM-TOP COMPILE-PICK 1 SH+! ;
: STORE ( locn ---)  -1 SH+! OFFSET-FROM-TOP  COMPILE-POST ;
: RESET ( locn ---)  DUP OFFSET-FROM-TOP COMPILE-DROPS SH! ;
: ADDTO ( locn ---)  -1 SH+! OFFSET-FROM-TOP  COMPILE (BUMP)  UNITS,  ;
```

Sec. 13. IN(

The stack height is maintained throughout compilation of a colon definition, but symbolic names can be used only within a section of code bounded by **IN(** and **OUT(**. The "stack height" within this section of code is considered to be the height above an imaginary stack frame located just below the first argument in the expression **IN(arguments)**. Initially this height is just the number of arguments and the stack height counter is set to this value.

```
VARIABLE UNUSED-HEIGHT  \ Unused as arguments for IN(
: IN(
   SH@ UNUSED-HEIGHT !  Ø SH!  BEGIN-SYMBOLIC SYMBOL-D'Y
   BEGIN  1 SH+!  SYMBOL IMMEDIATE  LAST @ @ NAME-)-IMMEDIATE =
   UNTIL  HIDE -1 SH+!  NORMAL-D'Y  SH@ NEGATE UNUSED-HEIGHT +!
   GAIN-WARNINGS @ GAIN-WARNINGS-SAVE !  GAIN-WARNINGS ON
   ; IMMEDIATE
```

Sec. 14. OUT(

OUT(must move each output value from its current location on the stack to the location required for output. No value will be moved to a higher location on the stack. First the numbver of initial outputs that are already in their desired position is determined, avoiding compilation of code that "moves" data to where it already is. If there are no remaining values, then the stacktop is simply reset. If there is just one remaining value and it is on the top of the stack, then it is moved by (NIPS), which is faster code for this common situation. Anything else is handled by (MOVES). This word takes its arguments from the parameter field. It expects an offset to the first location to be moved to, followed by a sequence of distances to be moved (one for each output value), followed by a terminating zero. The "distances" are precomputed to values convenient for the processor.

```
SYMBOL AAA  ' AAA @  FORGET AAA  CONSTANT SYMBOL-CODE
: ALREADY-THERE ( --- n )
   1 BEGIN  >IN @ OVER  ' DUP @ SYMBOL-CODE =  \ a symbol ?
         -ROT >BODY @ = AND        \ ... and there ?
    WHILE  DROP 1+
    REPEAT  >IN ! 1- ;

CREATE )  -1 ,  \ Not a legitimate stack location
: ")"-NEXT?  ( --- flag )
   >IN @  LOCATION'  SWAP >IN !  [ ' ) >BODY @ ] LITERAL = ;
: LAST&TOP?  ( --- flag )
   >IN @  LOCATION'  SH@ = ")"-NEXT? AND  SWAP >IN ! ;

CODE (NIPS)  DX POP AX LODS  AX SP ADD  DX PUSH  NEXT END-CODE
: NIP-OUT ( already-there --- )
   COMPILE (NIPS)  SH@ OVER - 1- UNITS,  1+ SH! ;

CODE (MOVES)  \ Expects: offset dist1 ... distN Ø  ( N >= 1 )
 SP DI MOV  AX LODS  AX DI ADD  AX LODS
 BEGIN  AX BX MOV  Ø [DI+BX] AX MOV  DI DEC  DI DEC
      AX Ø [DI] MOV  AX LODS  AX AX OR  Ø=
 UNTIL  DI SP MOV  NEXT END-CODE

: MOVE-OUT ( already there --- )
   COMPILE (MOVES)  DUP OFFSET-FROM-TOP UNITS,  SH!
   BEGIN  LOCATION' OFFSET-FROM-TOP
        DUP Ø>= ABORT" Output moved higher"
        UNITS,  1 SH+!  ")"-NEXT?
   UNTIL  Ø ,  ;
```

```
: OUT(   ALREADY-THERE ( n )
       LAST&TOP? IF ' ( last-arg ) DROP NIP-OUT
   ELSE ")"-NEXT?  IF RETOP
   ELSE           MOVE-OUT
   THEN THEN
   ' DROP ( right parenthesis )
   END-SYMBOLIC UNUSED-HEIGHT @ SH+!
   GAIN-WARNINGS-SAVE @ GAIN-WARNINGS ! ; IMMEDIATE
```

Sec. 15. Gain of New Colon Definitions

Because each offset from the stacktop is computed as a difference of stack heights, the stack height counter does not need to be initialized to any particular value. Colon initializes the stack height counter to zero so that at the end of the colon definition the counter's value will be exactly equal to the gain of the word. When a colon definition is completed, the gain of the newly defined word is stored in its gain field.

```
: : Ø SH!    VALID-HEIGHT ON : ;
: ; [COMPILE] ;
   HIGH? GAIN-WARNINGS @ AND
   IF .LAST ." has undefined gain " THEN
   SH@ GAIN ; IMMEDIATE
```