Quick Introduction to Forth
John J. Wavrik
Department of Mathematics
Univ of Calif – San Diego

The Forth Language

The base language used in this course is Forth. This is a computer language designed to create languages. Forth is easy to learn – it can be understood by understanding how it works! Unlike most conventional languages, its syntax is based on semantics: the emphasis is on what is done rather than on how it is said. The quickest way to introduce the reader to the process of programming is to examine and discuss some code. This is what we will do in the remainder of the article. At this point it should only be necessary to include some background information.

The Dictionary

The Forth language consists of a collection of commands. One thinks of these commands as words in a dictionary. The name for a command (which is called a "word") can be any collection of printable characters delimited by spaces. Each command has an associated action. The dictionary consists of the names and some stored representation of the action. The process of programming in Forth consists of adding new words (and actions) to the dictionary. A glossary of some important Forth words is found in Appendix A. Individual words can be accessed using the bookmarks in Adobe Reader.

Forth maintains its dictionary in chronological (rather than alphabetical) order. The dictionary looks like a linked list: when a new word is defined, it is added at the end of the dictionary.

<table>
<thead>
<tr>
<th>Previous Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
</tr>
</tbody>
</table>

Interpreting

The Forth system has two modes. The first (interpreting) occurs when the system reads a line of input. When a word is encountered in the input stream, the dictionary is searched (starting from the most recent entry). If the word is found, its action is performed. If a word is not found in the dictionary, the Forth system tries to interpret it as a number.
Every word has an action.
A word's action is performed when it is encountered.

Since a word's action is performed when the word is encountered, any parameters it needs for its action must be already available. Thus the words (or numbers) which are used as parameters must occur before the word in the input stream. Some means must be provided for storing these parameters. A stack is used for this. By convention, words act by removing their arguments from the stack and placing their results on the stack. (While numbers are not stored in the dictionary, they can be thought of as words whose action is to put themselves on the stack.)

Example:

```
3 2 +
```

This sequence (reading left to right) puts 3 on the stack, and puts 2 on the stack. + is a word which takes two numbers from the stack, adds them, and puts the result (5 in this case) on the stack. Here are some examples with two operations:

```
3 2 + 7 *
```

leaves 35 on the stack

while

```
3 2 7 * +
3 2 7 + *
```

leaves 17 on the stack

leaves 27 on the stack

There is no hierarchy of operations or use of parentheses to postpone operations. The sequence 3 2 7 puts these numbers on the stack (with 7 on top). The operations + and * are performed on the top two numbers on the stack.

NOTE: If you try these examples you will not see the results. The numbers on the stack are not usually displayed. Read on to find out how to see numbers on the stack.

The stack is a central feature of Forth. It is the means (rather than named variables) used by words to communicate with one another. In most languages data must be named before it can be used. A typical Forth word takes some unnamed parameters from the stack, performs some operation on them, and leaves the result(s) on the stack. The action of a word is usually documented by a "stack diagram" which shows what the word expects on the stack before it acts, and what it leaves after. Usually the programmer adds a further note of explanation (sometimes this is obvious from the name chosen for the word) The stack diagram for + and * is (n1 n2 n3) which indicates that they expect two integers on the stack and leave one integer. Their names indicate quite clearly what they are intended to do.
**Seeing what is on the stack**

If you type 3 2 + (and press enter) you will not see anything! The result is on the stack. The stack is not visible. There are two operations that you must know about for seeing results:

<table>
<thead>
<tr>
<th>.</th>
<th>( n – )</th>
<th>This is a dot (period). It prints the number on top of the stack (and removes it)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.S</td>
<td>( -- )</td>
<td>This (dot S) prints the stack without removing anything. The numbers are printed horizontally with the top of the stack on the right.</td>
</tr>
</tbody>
</table>

**Example:**

```
3 2 7 + .S [2] 3 9
```

We started with 3, 2 and 7 on the stack, but the top two numbers were added leaving 3 and 9. In FPC the number of elements on the stack is shown (by .S) in square brackets followed by the stack elements themselves.

```
* .S [1] 27
```

After multiplication .S shows that there is only 1 number on the stack and it is 27. In both cases .S does not remove numbers.

```
. 27
```

Finally we type . (dot) and the 27 is printed and removed from the stack.

There are several words in the Forth language whose action is to rearrange the items on the stack. Here are the basic ones:

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP</td>
<td>( n – n n ) duplicate number on top of stack</td>
</tr>
<tr>
<td>DROP</td>
<td>( n -- ) drop number on top of stack</td>
</tr>
<tr>
<td>SWAP</td>
<td>( n1 n2 – n2 n1 ) swap numbers on top of stack</td>
</tr>
<tr>
<td>ROT</td>
<td>( n1 n2 n3 – n2 n3 n1 ) bring 3rd number to top</td>
</tr>
<tr>
<td>OVER</td>
<td>( n1 n2 – n1 n2 n1 ) copy second element to top</td>
</tr>
</tbody>
</table>

To see the number on top of the stack without removing it, you can use **DUP .** (i.e. duplicate the number before printing it)
Compiling

Programming in Forth is equivalent to adding new words to the dictionary. Here is the main way new words are added. The Forth word \( : \) (colon) is used to start a definition – it is followed by the name of the word to be added. The definition is terminated by the Forth word \( ; \) (semicolon). The action of the \( : \) (colon) is to create a new dictionary header and switch the system to compiling mode. The action of \( ; \) (semicolon) is to compile a termination word and to return the system to the interpreting mode. The other words in the definition describe a sequence of actions to be taken when the new word executes. At the time of definition, these words do not carry out their action – the action is stored in the new definition.\(^1\)

Example:

\[
: 2* \text{DUP + ;}
\]

We have chosen to name the new word \( 2* \). The action of this word is to perform the action of \text{DUP} followed by the action of \text{+}. This adds the number on the stack to itself.

Compiling the action of component words results in a major speed increase without sacrificing the interactive nature of the language. The most time consuming step is the search of the dictionary. The new word \text{AddTest}:

\[
: \text{AddTest} 3 2 + \text{DROP ;}
\]

Executes 10,000,000 times in 1 second. The sequence of component words:

\[
3 2 + \text{DROP}
\]

Takes 600 seconds to evaluate 10,000,000 times. This increase of 600x in execution speed occurs because, in interpreting mode, the dictionary must be searched each time the sequence is executed. In the compiled word, the dictionary is only searched when \text{AddTest} is compiled.

Compilation is incremental: it occurs only when a new word is added and only affects the new definition. (It is neither desirable nor necessary to recompile an entire system to add a new command.)\(^2\) Indeed, new definitions can be made and compiled while in the midst of an interactive session. Execution speed for (compiled) Forth words is comparable to the execution speed for conventional compiled languages.

\(^1\) For the curious: Traditional Forth systems store the address (within the dictionary) of the component words. In some recent versions of Forth, words are subroutines in machine language and component words are compiled as subroutine calls. In either case, no dictionary search for component words is required when a word executes – the dictionary search takes place during compilation.

\(^2\) Modern Computer Algebra systems (like Maple and Mathematica) use a similar approach: they provide an extensible, interactive environment while retaining execution speed by compiling new procedures to an internal code.
**Terminology and Naming Convention**

The words : and ; function as if they were called something like "DEFINE" and "END-DEF".\(^3\) The definition of 2* might seem clearer if we were to write

<table>
<thead>
<tr>
<th>Define 2*</th>
<th>Make a new dictionary header with name 2*</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP +</td>
<td>body – describe the action of the new word</td>
</tr>
<tr>
<td>END-DEF</td>
<td>end the definition</td>
</tr>
</tbody>
</table>

Forth tends to use punctuation marks as names for frequently performed operations. Typing speed is not the only reason: these punctuation marks are often used as part of a naming convention to indicate what new words do. The words : and ; are often used by programmers to suggest opening and closing something: Win32Forth has words COMMENT: and COMMENT; which are used to open and close multi-line comments. The the word . (dot) used to print the number on the top of the stack is often used as part of the name of a word which prints something. A Forth programmer who sees a word with a dot in it will automatically assume that this word is intended to print. Later in this paper, for example, we will define a word "\texttt{Set.}" which (as the naming convention suggests) is a word which prints a set.

It is extremely helpful, when writing Forth programs, to choose names well. A good name should suggest its action. The use of programming conventions involving the punctuation marks is very helpful.

**Control Structures**

Forth provides control structures, such as IF .. THEN, which are like those found in most languages. IF and THEN are Forth words which are in the dictionary and, like other Forth words, they have associated actions. Most Forth words do not act during compilation – they are, instead, compiled into the current definition (as discussed above). Control structures change the flow of execution. In Forth, the control structures are implemented by words which carry out their action during compilation. The dictionary header for a word contains both the name of the word and some extra information about it. A word can be tagged (a so-called "immediate" word) so that its action is performed immediately, regardless of the state of the system. Because these words make changes in a dictionary entry, they are only used during compilation.

The placement of words like IF and THEN in sentences can be best understood if one knows their action. Let A, B and C be collections of words. We stipulate that A must leave a number on the stack. A zero is interpreted as FALSE, a non-zero number is

---

\(^3\) One can, in fact, easily add these new names to the dictionary and use them instead.
interpreted as TRUE. We do not want to execute A, B, C in sequence – we only wish to execute B if the number left by A is TRUE. A flow diagram for this behavior is:

```
A  ?  B  ->  C
```

Before B there must be a test of the number on the stack. If this number is 0 (FALSE) then we want to jump past B to the point right before C. Our source code must indicate where the test takes place and the target of the jump. The word IF compiles the conditional branching instruction and the word THEN indicates the target. Thus, in the definition of a new word, we write:

```
A  IF  B  THEN  C
```

This will be compiled into the dictionary entry of the word to look like this:

```
| code for A | ?BRANCH | target_addr | code for B | code for C |
```

The target address is right before the code for C. IF compiles the conditional branching instruction and leaves space for the target address (which it, of course, does not know at the moment) so it leaves on the stack the location of the missing address. The word THEN occurs after B. THEN knows where it is! It also knows (because IF left the information) where it must fill in the address of where it is. So THEN fills in the proper target address – and we are left with compiled code that looks like the above.

Notice that IF and THEN do not, themselves, occur in the dictionary entry of the word currently being defined (although we sometimes talk as if they do). Their job is to act during compilation. Their action is to leave behind code which will, when it executes, provides the proper control flow.

Note that the conditional branching instruction follows the convention of other Forth words: it removes the number tested (its argument) from the stack.

**Example:**

```
: ABS  DUP 0<  IF  NEGATE  THEN  ;
```

This word has the stack diagram ( n – n’ ) where n’ is the absolute value of n. The word 0< has the stack diagram ( n – t/f ) and NEGATE has the diagram ( n – n’ ) with n’ = -n. ABS negates a negative number but not a non-negative number. The definition can be written up with the stack comment and additional comments:

```
: ABS  ( n – n' ) \ n' is the negative of n
       DUP 0<  IF  NEGATE  THEN  ;
```
Besides IF .. THEN there are other control structures:

```
IF..ELSE..THEN
BEGIN..UNTIL
BEGIN..WHILE..REPEAT
DO..LOOP
```

Forth does not have a separate program called "the compiler" – the functionality of a conventional compiler is delegated to the action of the "immediate" words.

**Data**

The last thing that must be discussed in this brief introduction is how data is handled. The word : (colon) is called a defining word because it is used to add new words to the dictionary. There are other defining words which come with a Forth system (and Forth provides a mechanism for the user to make others). The defining words for data create a new word (a child) in the dictionary and allocate memory for the data associated with that word. They also specify the run-time action of their children. The child word usually will put either the address of the data or the data itself on the stack. For integer data we use three defining words: CONSTANT, VALUE, and VARIABLE. Here is how each of these creates a new word in the dictionary:

<table>
<thead>
<tr>
<th>Example</th>
<th>When new word is created:</th>
<th>When new word is used:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 CONSTANT xxx</td>
<td>Create a constant xxx initialized to 5</td>
<td>xxx puts 5 on the stack. A constant cannot be changed once it is defined.</td>
</tr>
<tr>
<td>5 VALUE yyy</td>
<td>Create a value yyy initialized to 5</td>
<td>yyy puts 5 on the stack but 7 TO yyy will change the stored value to 7.</td>
</tr>
<tr>
<td>VARIABLE zzz</td>
<td>Create a variable zzz (some implementations initialize to 0)</td>
<td>zzz puts the address of the storage on the stack. Words @ (fetch) and ! (store) can access and store data.</td>
</tr>
</tbody>
</table>

These are all defined using a more fundamental defining word CREATE. CREATE xxx will create a new dictionary entry with name xxx. When the word xxx is used, the address of the entry in memory will be put on the stack. CREATE is usually used with words like ALLOT (which sets aside storage for data).

*****
The Forth Language

Forth was invented during the 1970s by Charles Moore. Its original use was for applications involving hardware control and instrumentation. It was used as a high-level replacement for assembly language. Currently a major use of Forth is to program "embedded systems": computers which are included in other products. Forth has a long history of use in hardware applications.

In many respects the advantages of Forth for developing experimental hardware control systems are similar to the advantages we exploit for work in mathematics. Here are some of the attributes of Forth that make it a good choice for the kind of work we are discussing:

1. Forth is conceptually simple. It is fairly easy for a non-specialist to learn.
2. Forth is suited for creating special-purpose languages.
3. Forth provides an interactive computing environment.\(^4\)
   a. This speeds the development and testing of code.
   b. Programs can be easily written for interactive use.
4. Forth programming creates a dictionary of useful words. This fine-grained modularity improves clarity and contributes to producing reusable code.
5. It is easy to introduce new types of algebraic objects and operations seamlessly.
6. Forth is small and efficient.
7. Programs written in Forth execute at a speed comparable to conventional compiled languages.
8. Forth is extensible—extensibility includes introducing new data types and new language features.
9. Details of the language implementation are accessible to the user
10. Forth supports many programming styles and paradigms

Forth is widely available. An ANSI standard for the language was adopted in 1994. The core features of Forth have become quite stable. Implementations of the language for most platforms now offer the basic foundation of ANS-Forth.

We will explore a "middle-out" strategy for developing software: a skeleton system is created which is used to assist in developing progressively fuller systems. We exploit the fact that Forth allows refinements and extensions to be made without destroying the functionality of the underlying system.

\(^4\) By this we mean that a user's work with a system is a "session": commands can be issued from the keyboard to act on persistent data. The user can issue a command, see the result, and continue operations on the data. The systems produced by the methodology in this article also include the ability to create new commands during a session.
**IF and THEN**

This is really beyond the scope of the intended simple introduction. However, some may be curious about exactly how one can define words which do what we have asserted that IF and THEN do. Here are the simplest forms of the definitions of IF and THEN used in traditional Forth implementations:

```forth
: IF  COMPILE ?BRANCH  HERE 0 , ; IMMEDIATE
: THEN  HERE SWAP ! ; IMMEDIATE
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Start definition of IF</td>
</tr>
<tr>
<td>COMPILE ?BRANCH</td>
<td>Compile conditional branch instruction</td>
</tr>
<tr>
<td>HERE</td>
<td>Put current address of dictionary pointer on stack</td>
</tr>
<tr>
<td>0 ,</td>
<td>Compile a zero as a placeholder to be filled in by proper address.</td>
</tr>
<tr>
<td>;</td>
<td>End the definition</td>
</tr>
<tr>
<td>IMMEDIATE</td>
<td>Tag this as an immediate word</td>
</tr>
</tbody>
</table>

```forth
: THEN  
   HERE     
   SWAP     
   !        
   ;        
   IMMEDIATE 
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THEN</td>
<td>Start definition of THEN. Remember that IF left an address on the stack.</td>
</tr>
<tr>
<td>HERE</td>
<td>We put where we are now on the stack</td>
</tr>
<tr>
<td>SWAP</td>
<td>Put location of missing address on top of current address</td>
</tr>
<tr>
<td>!</td>
<td>Store current address.</td>
</tr>
<tr>
<td>;</td>
<td>End the definition</td>
</tr>
<tr>
<td>IMMEDIATE</td>
<td>Tag this as an immediate word</td>
</tr>
</tbody>
</table>

In most Forth implementations additional information is put on the stack to check that conditionals are properly paired (an IF with a THEN, etc.)
Appendix A

Glossary of major Forth words

`,

( -- xt ) <spaces>name<space>
Apostrophe (pronounced: "tic") Skip leading space delimiters. Parse name delimited by a space. Find name and return its execution address.

-( n1 n2 -- n3 )
subtract n2 from n1.

!( x addr -- )
Exclamation mark (pronounced "store") Store x at addr

( Parentheses are used for comments. The opening parenthesis is a Forth word which must be enclosed by blanks on either side. The action of this word is to skip all text until a closing parenthesis is found. The closing parenthesis is a delimiter. Some versions of Forth only allow comments to extend for one line.

* ( n1 n2 -- n3 )
multiply n1 by n2

*/( n1 n2 n3 -- n4 )
Multiply n1 by n2 producing the intermediate double-cell result d. Divide d by n3 giving the single-cell quotient n4.

, ( x -- )
Store x in the dictionary in next available cell

.* ( n -- )
(pronounced "dot") Display n as a signed number followed by a space.
follow by text to "."

This word will print all text following it up to a delimiting double quote ("), This word can only be used during compilation (it actually saves the text to be printed in the dictionary).

follow by text to ".(

This word will print all text following it up to a delimiting closing parenthesis. It does not store the text but prints it immediately. It can be used interpretively, outside of definitions, to print text when a file is loading.

.ID

( nfa -- )

print the name at the given name field address

.R

( n1 n2 -- )

Display n1 right aligned in a field n2 characters wide. If the number of characters required to display n1 is greater than n2, all digits are displayed with no leading spaces in a field as wide as necessary.

.S

( -- )

Print the contents of the stack without removing elements.

/ (n1 n2 -- q )

Divide n1 by n2, giving the single-cell quotient q. See the comment on /MOD

/MOD

(n1 n2 -- r q )

Divide n1 by n2, giving the single-cell remainder r and the single-cell quotient q. If n1 and n2 differ in sign, FPC will choose r to have the same sign as n2 (floored division).

? ( addr -- )

This is a shorthand for @ . (print the single cell number at the given address)
**?DO**

( n1 n2 -- )

This starts a loop (like DO) but will not enter the loop if n1=n2.

**?DUP**

( x -- 0 | x x )

Duplicate x if it is not zero.

@

( addr -- x )

(pronounced "fetch") x is the single-cell value stored at addr

+

( n1 n2 -- n3 )

add n2 to n1

+!

( n addr -- )

Add n to the single-cell number stored at addr

<

( n1 n2 -- t/f )

The flag is true if and only if n1 is less than n2

<>

( x1 x2 -- t/f )

True if and only if x1 is not the same, bit for bit, as x2

=

( x1 x2 -- t/f )

True if and only if x1 is the same, bit for bit, as x2

>

( n1 n2 -- t/f )

The flag is true if and only if n1 is bigger than n2

>=

( n1 n2 -- t/f )

The flag is true if and only if n1 is bigger than or equal to n2
>\(R\)
\((\ x -- )\)
Move \(x\) to the return stack

\(0<\)
\((\ n -- t/f )\)
True if and only if \(n\) is less than zero

\(0<>\)
\((\ n -- t/f )\)
True if and only if \(n\) is not equal to zero

\(0=\)
\((\ n -- t/f )\)
True if and only if \(n\) is equal to zero

\(0>\)
\((\ n -- t/f )\)
True if and only if \(n\) is greater than zero

\(1-\)
\((\ n1 -- n2 )\)
Subtract one from \(n1\) giving the difference \(n2\)

\(1+\)
\((\ n1 -- n2 )\)
Add one to \(n1\) giving the sum \(n2\)

\(2!\)
\((\ x1 x2 addr -- )\)
Store the cell pair at \(addr\) with \(x2\) at \(addr\) and \(x1\) at the next cell

\(2*\)
\((\ n1 -- n2 )\)
Multiply \(n1\) by 2 giving the product \(n2\)

\(2/\)
\((\ n1 -- n2 )\)
Divide \(n1\) by 2 giving the quotient \(n2\)
\(2@\)

( \text{addr} \rightarrow x1 \ x2 )

Fetch the cell pair stored at addr. \(x2\) is stored at addr and \(x1\) and the next consecutive cell.

\(2DROP\)

( \ x1 \ x2 \rightarrow \)

Drop the cell pair \(x1\) \(x2\) from the stack

\(2DUP\)

( \ x1 \ x2 \rightarrow \ x1 \ x2 \ x1 \ x2 )

Duplicate the cell pair \(x1\) \(x2\)

\(2SWAP\)

( \ x1 \ x2 \ x3 \ x4 \rightarrow \ x3 \ x4 \ x1 \ x2 )

Exchange the two top cell pairs

\(3DUP\)

( \ x1 \ x2 \ x3 \rightarrow \ x1 \ x2 \ x3 \ x1 \ x2 \ x3 )

Repeat the top cell triple

\(ABORT\)

( \ -- \)

Empty the stacks returning control to the outer interpreter

\(ABORT^\prime\)

( \ x1 \rightarrow \)

Abort\(^*\) <message>\(^*\) if any bit of \(x1\) is non-zero this will display the message and perform ABORT

\(ALLOT\)

( \ n \rightarrow \)

Move the dictionary pointer \(n\) units. If \(n\) is positive this reserves space in the dictionary. If \(n\) is negative it releases space.

\(AND\)

( \ x1 \ x2 \rightarrow \ x3 )

\(x3\) is the bit by bit logical and of \(x1\) with \(x2\)

\(ASCII\)

follow by string ( \ -- \ c )

c is the ASCII code of the first character in the string
**BEGIN**
Initiate BEGIN .. UNTIL or BEGIN .. WHILE .. REPEAT control structures

**BETWEEN**
( n a b -- t/f )
true if and only if a <= n <= b

**BL**
( -- n )
ASCII code for the space character (blank)

**BODY>**
( pfa -- cfa )
Move from the address of the body of a word to the execution address (code field address)

**C!**
( c addr -- )
Store byte c at addr

**C,**
( c -- )
Store the byte c in the next available dictionary location and increase the dictionary pointer

**C@**
( addr -- c )
Fetch the single byte, c, located at addr

**CELL**
( -- n )
n is the size, in bytes, of a cell

**CELL+**
( addr1 -- addr2 )
add the size of a cell to addr1 to get addr2

**CELLS**
( n1 -- n2 )
n2 is the size of n1 cells in bytes
**CMOVE**

( addr1 addr2 u -- )

Copy u consecutive characters from the block of memory starting at addr1 to the block of memory starting at addr2. The copying is character-by-character starting from the lower address to the higher.

**COMPARE**

( addr1 u1 addr2 u2 -- n )

Compare the string1 (addr1 with length u1) to string2 (addr2 with length u2). Imagine the shorter string to be padded with blanks to match the length of the longer. If string1 comes before string2, then n is -1, if the strings are equal then n=0, if string1 comes after string2 then n is 1.

**COUNT**

( cs-addr -- addr2 u )

cs-address is the address of a counted string. addr2 is the address of the characters and u is the count of characters.

**CR**

cause subsequent output to be at the start of the next line

**CREATE**

follow by spaces and a name

skip leading spaces and parse name up to a space delimiter. Make a dictionary entry for the name. CREATE xxx will make a new word xxx whose action will be to return the address of its body

**DECR**

( addr -- )

Decrement the integer stored at addr

**DEPTH**

( -- n )

n is the number of single cell values that were in the stack before n was placed on it

**DO**

( lim start -- )

used in DO ... LOOP control structure. This compiles a word which sets up limits for a loop. start is the starting index, the loop continues when the current index crosses the boundary between lim-1 and lim (the test occurs at LOOP)
DOES>

Used with CREATE to make a defining word for words with a family behavior. : <name> ... CREATE <words> DOES> <words> ; When this word is evoked it will make a new dictionary entry from the following word; the words between CREATE and DOES> describe what happens when the new word is compiled; The words after DOES> describe the action. Example: : CONS CREATE , DOES> @ ; defines a word which acts exactly like CONSTANT.

DROP

( x -- )
remove x from the stack

DUP

( x -- x x )
duplicate x

ELSE

Used in IF .. ELSE .. THEN construct. A IF B ELSE C THEN (where A,B,C are collections of words and A leaves a number on the stack) will execute B if the number on the stack is non-zero, C if it is zero (in either case execution continues after THEN)

EMIT

( c -- )
c is the ASCII code of a character (printable or control). EMIT displays the character

ERASE

( addr u -- )
set to zero u consecutive bytes starting at addr

EXECUTE

( cfa -- )
Remove the code address of a word from the stack and execute the word. At the keyboard this does the same as typing the word -- but this can be used to implement jump tables

EXIT

This immediately returns execution to the word from which the current word was called. Before exiting from a DO..LOOP the word UNLOOP must be used before EXIT to discard the loop parameters
**FALSE**

( -- 0 )
In Forth 0 is taken as FALSE and any non-zero value as TRUE. This word is used to make clear that a boolean is at issue.

**FILL**

( addr u char -- )
Store char in each of u consecutive memory locations beginning at addr

**HERE**

( -- addr )
addr is the position of the dictionary pointer

**I**

( -- n )
used within a DO .. LOOP construct, n is the current value of the loop index

**IF**

used in IF .. THEN and IF .. ELSE .. THEN this marks the location of a forward branch

**INCLUDE**

INCLUDE <filename>
read an interpret a file consisting of Forth code. This makes the file the input stream

**INCR**

( addr -- )
Increment the integer stored at addr

**IS**

<n> IS <value>
This has be changed to TO in the ANSI Standard. In FPC the word is IS IF X is a VALUE datatype, 3 IS X makes 3 the value of X

**J**

Used in a nested loop construct DO .. DO .. LOOP .. LOOP. Within the inner loop this is the current value of the index of the next outermost loop
**KEY**

( -- char )

Wait for a key press and return the corresponding character

**KEY?**

( -- t/f )

return TRUE if and only if a key has been pressed since the last operation that consumes key presses. This indicates that a key character is in the buffer but does not clear the buffer

**LEAVE**

Inside a DO .. LOOP construct this causes an immediate exit from the loop

**LOOP**

Completes a DO .. LOOP construction. This resolves the forward branch left by DO and compiles a word which tests the loop parameters for the exit condition

**MAX**

( n1 n2 -- n3 )

n3 is the larger of n1 and n2

**MIN**

( n1 n2 -- n3 )

n3 is the smaller of n1 and n2

**MOD**

( a b -- r )

\[ a = bq + r \] in floored division \( r \) has the same sign as \( b \), in centered division it has the same sign as \( a \).  

**MOVE**

( addr1 addr2 u -- )

Move \( u \) bytes from \( addr1 \) to \( addr2 \). After the move the \( u \) consecutive units starting at \( addr1 \) are exactly what was after \( addr1 \) before the move

**NEGATE**

( n -- n')

\( n' \) is -\( n \)
**NIP**

( x1 x2 -- x2 )

Drop the element below the top of the stack

**NOT**

( x1 -- x2 )

This is not in the ANSI standard because two meanings have been available. In some versions of Forth, NOT exchanges TRUE and FALSE (equivalent to 0=) in others it inverts the bits of x1 (if TRUE is represented by a number with all bits set, the latter does the former)

**NUMBER**

( cstr -- d )

Convert the counted string cstr to a double precision number

**NUMBER?**

( addr u -- d t/f )

Attempt to convert the string at addr of u characters to a number. TRUE if and only if the conversion is successful. d is the result, as double precision

**OFF**

( var -- )

Set the value of variable var to zero (FALSE)

**ON**

( var -- )

Set the value of variable var to -1 (TRUE)

**OR**

( x1 x2 -- x3 )

x3 is bitwise inclusive or of x1 with x2

**OVER**

( x1 x2 -- x1 x2 x1 )

Copy the second element on the stack to the top

**PAGE**

clear the display, return cursor to home position
PARSE
( char -- c-addr u )<text>
Read the following input text until a delimiter char is found. Store the
text in the input buffer. Return c-addr, the address where the text is
stored, and u the length of the parsed string.

PICK
( xu ... x1 x0 u -- xu ... x1 x0 xu )
Remove u. Copy the xu to the top of the stack.

PLACE
( addr len dest -- )
convert the string of len characters at addr to a counted string placed
at dest

QUERY
Gets input from the keyboard up to <ret> and places it into the
terminal input buffer, resetting the input buffer pointer

R@
( -- x ) (R: x -- x )
Copy the top of the return stack to the parameter stack

R>
( -- x ) (R: x -- )
Remove the element on top of the return stack and put it on the
parameter stack

RECURSE
When used in definition this compiles the action of the current
definition (a self-reference)

REPEAT
Used in BEGIN .. WHILE .. REPEAT. Control is sent back to point after
BEGIN

ROLL
( xu ... x1 x0 u -- xu-1 ... x1 x0 xu )
Remove u. Move the xu to the top of the stack and push other elements down.

ROT
( a b c -- b c a )
Rotate third stack element to top
**SPACE**

( -- )

Display one space

**SPACES**

( n -- )

Display n spaces

**STATE**

( -- addr )

addr is the address of a cell which contains the compilation state flag

**SWAP**

( a b -- b a )

Swap top two elements on stack

**TAB**

( -- )

Print spaces to get to the next TAB increment as specified by the variable TABSIZE.

**THEN**

Terminate IF..THEN construct, this sets target of conditional branch instruction compiled by IF

**TRUE**

( -- true )

Puts the value of a true flag on the stack (the integer with all bits set = -1 for most versions of Forth)

**TUCK**

( a b -- b a b )

tuck the top stack element under the element next below

**TYPE**

( addr cnt -- )

Print the string of cnt bytes starting at addr

**U<**

( u1 u2 -- t/f )

Interpret the top two stack elements as unsigned integers. Returns true if and only if u1 < u2
**U>**

( u1 u2 -- t/f )

Interpret the top two stack elements as unsigned integers. Returns true if and only if u1 > u2

**UM/MOD**

( ud u1 -- ur uq )

Divide the double precision number ud by u1 giving the remainder ur and quotient uq. All values and arithmetic are unsigned

**UNTIL**

Terminate **BEGIN** -- **UNTIL** construct. This compiles a conditional branch back to the place marked by **BEGIN**

**UPC**

( c -- c' )

c' is the character c converted to upper case (non-standard)

**UPPER**

( addr len -- )

convert the string of len characters at addr to upper case (non-standard)

**WHILE**

Use in **BEGIN** .. **WHILE** .. **REPEAT** to compile a conditional branch to place marked just after **REPEAT** if false

**WITHIN**

( n a b -- t/f )

TRUE if a <= n < b

**WORD**

<delim> WORD <text> -- addr

Skip leading delimiters. Parse the text up to delim. Store as a counted string in a transient area and return the address addr. Returns a string of length zero if no delim is found

**XOR**

( x1 x2 -- x3 )

x3 is the bit by bit exclusive or of x1 and x2

\Backslash is used for comments. It is a Forth word whose action is to discard all characters to the end of the line.