Forth Programming Language Cheatsheet

A concise reference for the Forth programming language, covering its core concepts, syntax, and common words for stack manipulation, arithmetic, control flow, and memory access.

Core Concepts and Stack Manipulation

Fundamental Principles

Stack-Based: Forth is a stack-based language where data is manipulated on a stack. Most operations involve pushing data onto the stack, performing operations on the top elements, and pushing the result back onto the stack.

Reverse Polish Notation (RPN): Forth uses RPN, also known as postfix notation, where operators follow their operands (e.g., `3 4 +` instead of `3 + 4`).

Words: Forth programs are built from 'words', which are essentially functions or commands. Words are executed in the order they appear.

Stack Manipulation Words

`DUP`	Duplicates the top item on the stack. **Example:** `10 DUP` (Stack: 10 -> 10 10)
`DROP`	Removes the top item from the stack. **Example:** `10 DROP` (Stack: 10 ->)
`SWAP`	Exchanges the top two items on the stack. **Example:** `10 20 SWAP` (Stack: 10 20 -> 20 10)
`OVER`	Duplicates the second item on the stack and places it on top. **Example:** `10 20 OVER` (Stack: 10 20 -> 10 20 10)
`ROT`	Rotates the top three items on the stack, bringing the third item to the top. **Example:** `10 20 30 ROT` (Stack: 10 20 30 -> 20 30 10)
`2DUP`	Duplicates the top two items on the stack. **Example:** `10 20 2DUP` (Stack: 10 20 -> 10 20 10 20)

Arithmetic and Logical Operations

Arithmetic Words

`+`	Adds the top two numbers on the stack. **Example:** `3 4 +` (Stack: -> 7)
`-`	Subtracts the top number from the second number on the stack. **Example:** `7 3 -` (Stack: -> 4)
`*`	Multiplies the top two numbers on the stack. **Example:** `4 5 *` (Stack: -> 20)
`\`	Divides the second number on the stack by the top number, returning the quotient. **Example:** `20 4 \` (Stack: -> 5)
`MOD`	Divides the second number on the stack by the top number, returning the remainder. **Example:** `22 5 MOD` (Stack: -> 2)
`/MOD`	Divides the second number by the top number, returning both the quotient and the remainder (remainder on top). **Example:** `22 5 /MOD` (Stack: -> 2 4)

Logical and Comparison Words

`=`	Compares the top two numbers on the stack for equality. Returns `TRUE` (-1) if equal, `FALSE` (0) otherwise. **Example:** `5 5 =` (Stack: -> -1) `5 6 =` (Stack: -> 0)
`<>` or `<>`	Compares the top two numbers for inequality. Returns `TRUE` (-1) if not equal, `FALSE` (0) otherwise. **Example:** `5 6 <>` (Stack: -> -1) `5 5 <>` (Stack: -> 0)
`<`	Compares if the second number on the stack is less than the top number. Returns `TRUE` (-1) if true, `FALSE` (0) otherwise. **Example:** `5 10 <` (Stack: -> -1) `10 5 <` (Stack: -> 0)
`>`	Compares if the second number on the stack is greater than the top number. Returns `TRUE` (-1) if true, `FALSE` (0) otherwise. **Example:** `10 5 >` (Stack: -> -1) `5 10 >` (Stack: -> 0)
`AND`	Performs a bitwise AND operation on the top two numbers. **Example:** `3 6 AND` (Stack: -> 2)
`OR`	Performs a bitwise OR operation on the top two numbers. **Example:** `3 6 OR` (Stack: -> 7)
`NOT`	Performs a bitwise NOT operation on the top number. **Example:** `0 NOT` (Stack: -> -1)

Control Flow and Definitions

Control Flow Structures

`IF ... THEN`: Conditional execution.

Example:

```
5 0 > IF  ." Greater than zero " THEN
```

`IF ... ELSE ... THEN`: Conditional execution with alternative.

Example:

```
5 0 < IF  ." Less than zero " ELSE  ." Not less than zero " THEN
```

`BEGIN ... UNTIL`: Looping structure that executes until a condition is true.

Example:

```
: COUNTDOWN  10 BEGIN  DUP .  1 -  DUP 0 = UNTIL  DROP ;
COUNTDOWN
```

`BEGIN ... WHILE ... REPEAT`: Looping structure that executes while a condition is true.

Example:

```
: STARS  BEGIN  DUP 0 > WHILE  DUP .  1 -  REPEAT  DROP ;
5 STARS
```

Defining New Words

`:` **(colon):** Starts the definition of a new word.
`;` **(semicolon):** Ends the definition of a new word.

Example:

```
: SQUARE  DUP DUP * ;
```

This defines a new word `SQUARE` that duplicates the top of the stack and multiplies the two copies, effectively squaring the number.

Defining constants and variables:

`CONSTANT` **:** Defines a constant.

Example:

```
10 CONSTANT TEN
TEN .  (Output: 10)
```

`VARIABLE` **:** Defines a variable (a memory location).

Example:

```
VARIABLE X
15 X !
X @ .  (Output: 15)
```

`!` is used to store a value into the variable and `@` is used to fetch the value from the variable.

Memory Access and I/O

Memory Access Words

`!` (store)	Stores a value at a specified memory address. The address is on top of the stack, followed by the value to store. **Example:** `1000 25 !` (Stores the value 25 at memory address 1000)
`@` (fetch)	Fetches the value from a specified memory address and places it on the stack. The address is on top of the stack. **Example:** `1000 @` (Fetches the value from memory address 1000)
`+!` (add store)	Adds a value to the content of a specified memory address. The address is on top of the stack, followed by the increment value. **Example:** `1000 5 +!` (Adds 5 to the value at memory address 1000)
`C!` (c-store)	Stores a byte at a specified memory address. **Example:** `1000 65 C!`
`C@` (c-fetch)	Fetches a byte from a specified memory address. **Example:** `1000 C@`

Input/Output Words

`.` (dot)	Prints the top number on the stack to the console, followed by a space. **Example:** `42 .` (Output: 42)
`.S`	Prints the contents of the data stack without modifying it (stack snapshot). **Example:** `1 2 3 .S` (Output: <1> 1 <2> 2 <3> 3)
`EMIT`	Prints the character corresponding to the ASCII value on top of the stack. **Example:** `65 EMIT` (Output: A)
`."` (dot-quote)	Prints a string literal to the console. The string is enclosed in double quotes. **Example:** `." Hello, Forth! "` (Output: Hello, Forth!)
`KEY`	Reads a character from the input and places its ASCII value on the stack. **Example:** `KEY .` (Waits for a key press, then prints its ASCII value)