Easy Forth

by Nick Morgan

View on GitHub Page PDF
Introduction 2
Adding Some Numbers 2
Defining Words 5
Stack Manipulation 6
Generating Output 9
Conditionals and Loops 12
Variables and Constants 18
Arrays 21
Keyboard Input 22
added Debugger 24
Snake! 26
The End 39
added Forth Word List 40
added sort alphabetically 42
Canvas example 24x24 44

This is a brilliant Forth Learning Documentation.

Including a limited Forth implementation in Javascript online, so no download needed to try some Forth,

and the Javascript software written to be found on github.

Here just with some added explanations for reading, some links from a word list, and for print

plus the added Game Canvas area for easier understanding of the software example,

added by Juergen Pintaske, ExMark as v2 (work in progress)
And to see what is happening inside a small Debugger

Thanks for the good work Nick, adding a very nice learning resource for anybody who wants to try Forth.

Easy Forth v16_A5_withexp_comments

Introduction

This small ebook is here to teach you a programming language called Forth. Forth is a language unlike most others. It's not functional *or* object oriented, it doesn't have type-checking, and it basically has zero syntax. It was written in the 70s, but is still used today for certain applications.

Why would you want to learn such an odd language? Every new programming language you learn helps you think about problems in new ways. Forth is very easy to learn, but it requires you to think in a different way than you're used to. That makes it a perfect language to broaden your coding horizons.

This book includes a simple implementation of Forth I wrote in JavaScript. It's by no means perfect, and is missing a lot of the functionality you'd expect in a real Forth system. It's just here to give you an easy way to try out the examples. (If you're a Forth expert, please contribute here and make it better!)

I'm going to assume that you know at least one other programming language, and have a basic idea of how stacks work as a data structure.

Adding Some Numbers

The thing that separates Forth from most other languages is its use of the stack. In Forth, everything revolves around the stack. Any time you type a number, it gets pushed onto the stack. If you want to add two numbers together, typing `+` takes the top two numbers off the stack, adds them, and puts the result back on the stack.

Let's take a look at an example. Type (don't copy-paste) the following into the interpreter, typing `Enter` after each line.

```
1
2
3
```

<- Top

Every time you type a line followed by the `Enter` key, the Forth interpreter executes that line, and appends the string `ok` to let you know there were no errors. You should also notice that as you execute each line, the area at the top fills up with numbers. That area is our visualization of the stack. It should look like this:

```
1 2 3 <- Top
```

Now, into the same interpreter, type a single

```
+ ( 1 )
```

followed by the `Enter` key. The top two elements on the stack, `2` and `3`, have been replaced by `5`.

```
1 5 <- Top
```

At this point, your editor window should look like this:

```
1 ok 2 ok 3 ok + ok
```

Type `+` again and press `Enter`, and the top two elements will be replaced by 6. If you type `+` one more time, Forth will try to pop the top two elements off the stack, even though there's only *one* element on the stack! This results in a `Stack underflow` error:

```
1 ok 2 ok 3 ok + ok + ok + Stack underflow
```

Forth doesn't force you to type every token as a separate line. Type the following into the next editor, followed by the `Enter` key:

```
123 456 +
```

<- Top

The stack should now look like this:

```
579 <- Top
```

This style, where the operator appears after the operands, is known as Reverse-Polish notation. Let's try something a bit more complicated, and calculate `10 * (5 + 2)`. Type the following into the interpreter:

***   (2)**

```
5 2 + 10 *
```

<- Top

One of the nice things about Forth is that the order of operations is completely based on their order in the program. For example, when executing `5 2 + 10 *`, the interpreter pushes 5 to the stack, then 2, then adds them and pushes the resulting 7, then pushes 10 to the stack, then multiplies 7 and 10. Because of this, there's no need for parentheses to group operators with lower precedence.

Most Forth words affect the stack in some way. Some take values off the stack, some leave new values on the stack, and some do a mixture of both. These "stack effects" are commonly represented using comments of the form

(before -- after).

For example, `+` is `( n1 n2 -- sum )` - `n1` and `n2` are the top two numbers on the stack, and `sum` is the value left on the stack.

Defining Words

The syntax of Forth is extremely straightforward. Forth code is interpreted as a series of space-delimited words. Almost all non-whitespace characters are valid in words. When the Forth interpreter reads a word, it checks to see if a definition exists in an internal structure known as the Dictionary. If it is found, that definition is executed. Otherwise, the word is assumed to be a number, and it is pushed onto the stack. If the word cannot be converted to a number, an error occurs.

You can try that out yourself below. Type `foo` (an unrecognized word) and press enter.

<- Top

You should see something like this:

```
foo foo ?
```

`foo ?` means that Forth was unable to find a definition for `foo`, and it wasn't a valid number.

We can create our own definition of `foo` using two special words called

`: (colon) and ; (semicolon). :  ( 3, 4 )`

is our way of telling Forth we want to create a definition. The first word after the `:` becomes the definition name, and the rest of the words (until the `;`) make up the body of the definition. It's conventional to include

two spaces between the name and the body of the definition. Try entering the following:

```
: foo  100 + ;
1000 foo
foo foo foo
```

Warning: A common mistake is to miss out the space before the `;` word. Because Forth words are space delimited and can contain most characters, `+;` is a perfectly valid word and is not parsed as two separate words.

<- Top

As you've hopefully figured out, our `foo` word simply adds 100 to the value on top of the stack. It's not very interesting, but it should give you an idea of how simple definitions work.

Stack Manipulation

Now we can start taking a look at some of Forth's predefined words. First, let's look at some words for manipulating the elements at the top of the stack.

`dup ( n -- n n ) ( 5 )`

`dup` is short for "duplicate" – it duplicates the top element of the stack. For example, try this out:

```
1 2 3 dup
```

<- Top

You should end up with the following stack:

1 2 3 3 <- Top

drop (n --)   (6)

drop simply drops the top element of the stack. Running:

```
1 2 3 drop
```

gives you a stack of:

1 2 <- Top

<- Top

swap (n1 n2 -- n2 n1)   (7)

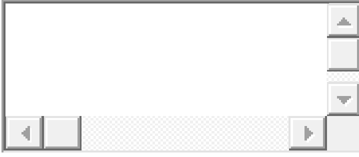
swap, as you may have guessed, swaps the top two elements of the stack. For example:

```
1 2 3 4 swap
```

will give you:

1 2 4 3 <- Top

<- Top



over (n1 n2 -- n1 n2 n1) (8)

`over` is a bit less obvious: it takes the second element from the top of the stack and duplicates it to the top of the stack. Running this:

```
1 2 3 over
```

will result in this:

```
1 2 3 2 <- Top
```

<- Top

rot (n1 n2 n3 -- n2 n3 n1) (9)

Finally, `rot` "rotates" the top *three* elements of the stack. The third element from the top of the stack gets moved to the top of the stack, pushing the other two elements down.

```
1 2 3 rot
```

gives you:

2 3 1 <- Top

<- Top

Generating Output

Next, let's look at some words for outputting text to the console.

`. ( n -- ) (period)   ( 10 )`

The simplest output word in Forth is `.`. You can use `.` to output the top of the stack in the output of the current line. For example, try running this (make sure to include all the spaces!):

```
1 . 2 . 3 . 4 5 6 . . .
```

<- Top

You should see this:

```
1 . 2 . 3 . 4 5 6 . . . 1 2 3 6 5 4 ok
```

Going through this in order, we push `1`, then pop it off and output it. Then we do the same with `2` and `3`. Next we push `4`, `5`, and `6` onto the stack. We then pop them off and output them one-by-one. That's why the last three numbers in the output are reversed: the stack is last in, first out.

emit (c --) (11)

`emit` can be used to output numbers as ascii characters. Just like `.` outputs the number at the top of the stack, `emit` outputs that number as an ascii character. For example:

```
 33 119 111 87 emit emit emit emit
```

<- Top

I won't give the output here so as to not ruin the surprise. This could also be written as:

```
87 emit 111 emit 119 emit 33 emit
```

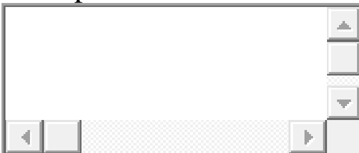
Unlike `.`, `emit` doesn't output any space after each character, enabling you to build up arbitrary strings of output.

cr (--) (12)

`cr` is short for carriage return – it simply outputs a newline:

```
cr 100 . cr 200 . cr 300 .
```

<- Top



This will output:

```
cr 100 . cr 200 . cr 300 . 100 200 300 ok
```

`." ( -- )   ( 13 )`

Finally we have `."` – a special word for outputting strings. The `."` word works differently inside definitions to interactive mode. `."` marks the beginning of a string to output, and the end of the string is marked by `"`. The closing `"` isn't a word, and so doesn't need to be space-delimited. Here's an example:

```
: say-hello  ." Hello there!" ;
say-hello
```

<- Top

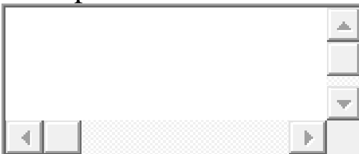
You should see the following output

say-hello Hello there! ok

We can combine `."`, `.`, `cr`, and `emit` to build up more complex output:

```
: print-stack-top  cr dup ." The top of the stack is " .
  cr ." which looks like '" dup emit ." ' in ascii  " ;
48 print-stack-top
```

<- Top



Running this should give you the following output:

48 print-stack-top The top of the stack is 48 which looks like '0' in ascii ok

Conditionals and Loops

Now onto the fun stuff! Forth, like most other languages, has conditionals and loops for controlling the flow of your program. To understand how they work, however, first we need to understand booleans in Forth.

Booleans (14) (0 = false)

There's actually no boolean type in Forth. The number `0` is treated as false, and any other number is true, although the canonical true value is `-1` (all boolean operators return `0` or `-1`).

To test if two numbers are equal, you can use

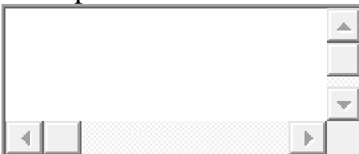
= (15)

```
3 4 = .
5 5 = .
```

This should output:

```
3 4 = . 0 ok 5 5 = . -1 ok
```

<- Top



You can use < and > for less than and greater than. < checks to see if the second item from the top of the stack is less than the top item of the stack, and vice versa for >:

< (16)

> (17)

```
3 4 < .
3 4 > .
3 4 < . -1 ok 3 4 > . 0 ok
```

<- Top

The boolean operators And, Or, and Not are available as `and`, `or`, and `invert`:

AND (18)

OR (19)

INVERT (20)

```
3 4 < 20 30 < and .
3 4 < 20 30 > or .
3 4 < invert .
```

The first line is the equivalent of `3 < 4 & 20 < 30` in a C-based language. The second line is the equivalent of `3 < 4 | 20 > 30`. The third line is the equivalent of `!(3 < 4)`.

`and`, `or`, and `invert` are all bitwise operations. For well-formed flags (`0` and `-1`) they'll work as expected, but they'll give incorrect results for arbitrary numbers.

<- Top

if then (21) (23)

Now we can finally get onto conditionals. Conditionals in Forth can only be used inside definitions. The simplest conditional statement in Forth is `if then`, which is equivalent to a standard `if` statement in most languages. Here's an example of a definition using `if then`. In this example, we're also using the `mod` word, which returns the modulo of the top two numbers on the stack. In this case, the top number is 5, and the other is whatever was placed on the stack before calling `buzz?`. Therefore, `5 mod 0 =` is a boolean expression that checks to see if the top of the stack is divisible by 5.

```
: buzz?  5 mod 0 = if ." Buzz" then ;
3 buzz?
4 buzz?
5 buzz?
```

<- Top

This will output:

```
3 buzz? ok 4 buzz? ok 5 buzz? Buzz ok
```

It's important to note that the `then` word marks the end of the `if` statement. This makes it equivalent to `fi` in Bash or `end` in Ruby, for example.

Another important thing to realize is that `if` consumes the top value on the stack when it checks to see if it's true or false.

if else then (22)

`if else then` is equivalent to an `if/else` statement in most languages. Here's an example of its use:

```
: is-it-zero?  0 = if ." Yes!" else ." No!" then ;
0 is-it-zero?
1 is-it-zero?
2 is-it-zero?
```

<- Top

This outputs:

```
0 is-it-zero? Yes! ok 1 is-it-zero? No! ok 2 is-it-zero?
No! ok
```

This time, the if clause (consequent) is everything between `if` and `else`, and the else clause (alternative) is everything between `else` and `then`.

do loop (24) (25)

`do loop` in Forth most closely resembles a `for` loop in most C-based languages. In the body of a `do loop`, the special word `i` pushes the current loop index onto the stack.

The top two values on the stack give the starting value (inclusive) and ending value (exclusive) for the `i` value. The starting value is taken from the top of the stack. Here's an example:

```
: loop-test  10 0 do i . loop ;
loop-test
```

<- Top

This should output:

loop-test 0 1 2 3 4 5 6 7 8 9 ok

The expression `10 0 do i . loop` is roughly equivalent to:

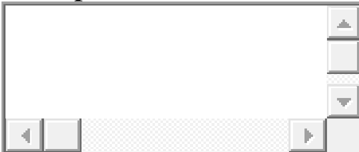
```
for (int i = 0; i < 10; i++) {
  print(i);
}
```

Fizz Buzz

We can write the classic Fizz Buzz program easily using a `do loop`:

```
: fizz?  3 mod 0 = dup if ." Fizz" then ;
: buzz?  5 mod 0 = dup if ." Buzz" then ;
: fizz-buzz?  dup fizz? swap buzz? or invert ;
: do-fizz-buzz  25 1 do cr i fizz-buzz? if i . then loop ;
do-fizz-buzz
```

<- Top



`fizz?` checks to see if the top of the stack is divisible by 3 using `3 mod 0 =`. It then uses `dup` to duplicate this result. The top copy of the value is consumed by `if`. The second copy is left on the stack and acts as the return value of `fizz?`.

If the number on top of the stack is divisible by 3, the string `"Fizz"` will be output, otherwise there will be no output.

`buzz?` does the same thing but with 5, and outputs the string `"Buzz"`.

`fizz-buzz?` calls `dup` to duplicate the value on top of the stack, then calls `fizz?`, converting the top copy into a boolean. After this, the top of the stack consists of the original value, and the boolean returned by `fizz?`. `swap` swaps these, so the original top-of-stack value is back on top, and the boolean is underneath. Next we call `buzz?`, which replaces the top-of-stack value with a boolean flag. Now the top two values on the stack are booleans representing whether the number was divisible by 3 or 5. After this, we call `or` to see if either of these is true, and `invert` to negate this value. Logically, the body of `fizz-buzz?` is equivalent to:

```
!(x % 3 == 0 || x % 5 == 0)
```

Therefore, `fizz-buzz?` returns a boolean indicating if the argument is not divisible by 3 or 5, and thus should be printed. Finally, `do-fizz-buzz` loops from 1 to 25, calling `fizz-buzz?` on `i`, and outputting `i` if `fizz-buzz?` returns true.

If you're having trouble figuring out what's going on inside `fizz-buzz?`, the example below might help you to understand how it works. All we're doing here is executing each word of the definition of `fizz-buzz?` on a separate line. As you execute each line, watch the stack to see how it changes:

```
: fizz?  3 mod 0 = dup if ." Fizz" then ;
: buzz?  5 mod 0 = dup if ." Buzz" then ;
4
dup
fizz?
swap
buzz?
or
invert
```

<- Top

Here's how each line affects the stack:

```
4          4 <- Top
dup        4 4 <- Top
fizz?      4 0 <- Top
swap       0 4 <- Top
buzz?      0 0 <- Top
or         0 <- Top
invert     -1 <- Top
```

Remember, the final value on the stack is the return value of the `fizz-buzz?` word. In this case, it's true, because the number was not divisible by 3 or 5, and so *should* be printed.

Here's the same thing but starting with 5:

```
5          5 <- Top
dup        5 5 <- Top
fizz?      5 0 <- Top
swap       0 5 <- Top
buzz?      0 -1 <- Top
or         -1 <- Top
invert     0 <- Top
```

In this case the original top-of-stack value was divisible by 5, so nothing should be printed.

Variables and Constants

Forth also allows you to save values in variables and constants. Variables allow you to keep track of changing values without having to store them on the stack. Constants give you a simple way to refer to a value that won't change.

Variables

Because the role of local variables is generally played by the stack, variables in Forth are used more to store state that may be needed across multiple words.

Defining variables is simple:

```
variable balance   ( 26 )
```

This basically associates a particular memory location with the name `balance`. `balance` is now a word, and all it does is to push its memory location onto the stack:

```
variable balance
balance
```

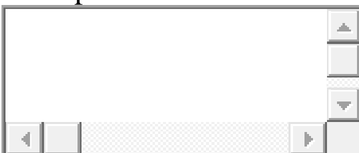
<- Top

You should see the value `1000` on the stack. This Forth implementation arbitrarily starts storing variables at the memory location `1000`.

The word `!` stores a value at the memory location referenced by a variable, and the word `@` fetches the value from a memory location:

```
variable balance
123 balance !
balance @
```

<- Top



This time you should see the value `123` on the stack. `123 balance` pushes the value and the memory location onto the stack, and `!` stores that value at that memory location. Likewise, `@` retrieves the value based on the memory location, and pushes that value onto the stack. If you've used C or C++, you can think of `balance` as a pointer that is dereferenced by `@`.

The word `?` is defined as `@ .` and it prints the current value of a variable. The word `+!` is used to increase the value of a variable by a certain amount (like `+=` in C-based languages).

```
variable balance
123 balance !
balance ?
50 balance +!
balance ?
```

<- Top

Run this code and you should see:

variable balance ok 123 balance ! ok balance ? 123 ok 50 balance +! ok balance ? 173 ok

Constants

If you have a value that doesn't change, you can store it as a constant. Constants are defined in one line, like this:

```
42 constant answer   ( 27 )
```

This creates a new constant called `answer` with the value `42`. Unlike variables, constants just represent values, rather than memory locations, so there's no need to use `@`.

```
42 constant answer
2 answer *
```

<- Top

Running this will push the value `84` on the stack. `answer` is treated as if it was the number it represents (just like constants and variables in other languages).

Arrays

Forth doesn't exactly support arrays, but it does allow you to allocate a zone of contiguous memory, a lot like arrays in C. To allocate this memory, use the `allot` word.

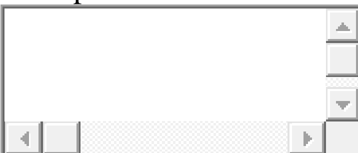
Cells (28)

Allot (29)

! (30)

```
variable numbers
3 cells allot
10 numbers 0 cells + !
20 numbers 1 cells + !
30 numbers 2 cells + !
40 numbers 3 cells + !
```

<- Top



This example creates a memory location called `numbers`, and reserves three extra memory cells after this location, giving a total of four memory cells. (`cells` just multiplies by the cell-width, which is 1 in this implementation.)

`numbers 0 +` gives the address of the first cell in the array. `10 numbers 0 + !` stores the value `10` in the first cell of the array.

We can easily write words to simplify array access:

```
variable numbers
3 cells allot
: number  ( offset -- addr )  cells numbers + ;

10 0 number !
20 1 number !
30 2 number !
40 3 number !
```

? (31)

```
2 number ?
```

<- Top

`number` takes an offset into `numbers` and returns the memory address at that offset. `30 2 number !` stores `30` at offset `2` in `numbers`, and `2 number ?` prints the value at offset `2` in `numbers`.

Keyboard Input

Forth has a special word called `key`, which is used for accepting keyboard input. When the `key` word is executed, execution is paused until a key is pressed. Once a key is pressed, the key code of that key is pushed onto the stack. Try out the following:

key (32)

```
key . key . key .
```

<- Top

When you run this line, you'll notice that at first nothing happens. This is because the interpreter is waiting for your keyboard input. Try hitting the `A` key, and you should see the keycode for that key, `65`, appear as output on the current line. Now hit `B`, then `C`, and you should see the following:

```
key . key . key . 65 66 67 ok
```

Printing keys with

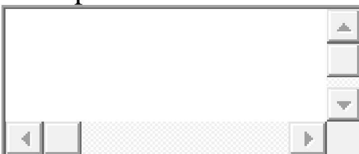
begin until (33) (34)

Forth has another kind of loop called `begin until`. This works like a `while` loop in C-based languages. Every time the word `until` is hit, the interpreter checks to see if the top of the stack is non-zero (true). If it is, it jumps back to the matching `begin`. If not, execution continues.

Here's an example of using `begin until` to print key codes:

```
: print-keycode  begin key dup . 32 = until ;
print-keycode
```

<- Top



This will keep printing key codes until you press space. You should see something like this:
print-keycode 80 82 73 78 84 189 75 69 89 67 79 68 69 32 ok

`key` waits for key input, then `dup` duplicates the keycode from `key`. We then use `.` to output the top copy of the keycode, and `32 =` to check to see if the keycode is equal to 32. If it is, we break out of the loop, otherwise we loop back to `begin`.

A Debugger – adaptable to your own needs

(added by Juergen Pintaske to these pages, fine with VFX and VFXTESTAPP.exe – here still an issue – has to be tested again.)

Seeing what is happening is the most important – especially if something goes wrong. There are 2 words in EASYforth that have not been described yet and can be used:

>R (to R)	Take the top value on the and transfer it to the	Data Stack Return Stack
R> (from R to D)	Take the value on the and move it to the	Return Stack Data Stack

And now we can understand how this little Debugger works – all data is displayed on one line:

First: Define a word ?????,

Send 3 Spaces,
Display 6 Variables TV1 TV2 TV3, TV4 TV5 TV6
and 4 spaces after it
then transfer the top 3 stack values on the Return Stack to the Data Stack using R>
and then each time DUPlicate the value on the Data Stack, display one, put the second
one back onto the RSTACK
with a Space in between
Send 4 Spaces after these 3 values
Now transfer the top 8 values on the Data Stack to the Return Stack
And do the same trick as before:
Fronm R to D, DUPlicate, print one, leave other on the Data Stack where it was before

The same for all 8 values that had been moved from the DStack to the RStack.

The Debugger came from another applicastion as

The Debugger ?????

```
: ?????  3 spaces tvar1 @ . tvar2 @ . tvar3 @ . tvar4 @ .
4 spaces    R>  R>  R>  DUP >R . Space DUP >R . space  DUP >R .
4 SPACES    >R >R >R >R >R >R >R >R       DUP . R> DUP . R> DUP .
R> DUP . R> DUP . R> DUP .  R> DUP . R> DUP . ;
```

One additional trick is to move FFFF as first value to the RSTACK and to the DStack to indicate the bottom position when the Debugger is called.

And now adapt this general Debugger to use the variables in EASYFORTH by copying the variables in SNAKE to show in this Debugger and have the 6 Variables displayed used in SNAKE

```
( copy the SNAKE Variables now and put 6 variables to
be displayed )
( snake-x-head   snake-y-head   apple-x   apple-y
direction   length )

: ??  3 spaces TV1 @ . TV2 @ . TV3 @ . TV4 @ . TV5 @
. TV6 @ .           4 spaces R> R> R> DUP >R . Space
DUP >R . space DUP >R .                   4 SPACES >R
>R >R >R >R >R >R >R    R> DUP . R> DUP . R> DUP . R>
DUP . R> DUP . R> DUP . R> DUP . R> DUP . ;


( into )

: ??  3 spaces    snake-x-head @ .   snake-y-head .
apple-x @ .   apple-y @ .   direction @ .    length @
.           4 spaces            R> R> R>   DUP >R .
Space DUP >R . space DUP >R .
4 SPACES    >R >R >R >R >R >R >R >R    R> DUP . R>
DUP . R> DUP . R> DUP . R> DUP . R> DUP . R> DUP . R>
DUP . ;


\ And as I have found out, it breaks easyFORTH
```

Snake!

Now it's time to put it all together and make a game! Rather than having you type all the code, I've pre-loaded it into the editor.

Before we look at the code, try playing the game. To start the game, execute the word `start`. Then use the arrow keys to move the snake. If you lose, you can run `start` again.

<- Top

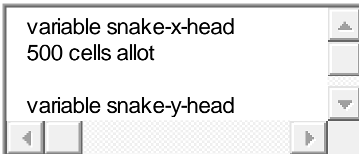
```
variable snake-x-head ok
500 cells allot ok
Ok

variable snake-y-head ok
500 cells allot ok
Ok

variable apple-x ok
variable apple-y ok
ok

0 constant left ok
1 constant up ok
2 constant right ok
3 constant down ok
Ok

24 constant width ok
24 constant height ok
Ok

variable direction ok
variable length ok
ok

: snake-x ( offset -- address )
cells snake-x-head + ; ok
ok

: snake-y ( offset -- address )
cells snake-y-head + ; ok
ok
```

```
: convert-x-y ( x y -- offset ) 24 cells * + ; ok
: draw ( color x y -- ) convert-x-y graphics + ! ; ok
: draw-white ( x y -- ) 1 rot rot draw ; ok
: draw-black ( x y -- ) 0 rot rot draw ; ok
Ok

: draw-walls
   width 0
   do
     i 0 draw-black
     i height 1 - draw-black
   loop
   height 0
   do
     0 i draw-black
     width 1 - i draw-black
   loop
; ok
Ok

: initialize-snake
   4 length !
   length @ 1 + 0
   do
      12 i - i snake-x !
      12 i snake-y !
   loop
   right direction ! ; ok
ok

: set-apple-position apple-x ! apple-y ! ; ok
Ok

: initialize-apple 4 4 set-apple-position ; ok
Ok

: initialize
  width 0
  do
    height 0
    do
      j i draw-white
    loop
  loop
  draw-walls
  initialize-snake
  initialize-apple ; ok
```

```
ok

: move-up -1 snake-y-head +! ; ok
: move-left -1 snake-x-head +! ; ok
: move-down 1 snake-y-head +! ; ok
: move-right 1 snake-x-head +! ; ok
Ok

: move-snake-head direction @
  left over = if move-left else
  up over = if move-up else
  right over = if move-right else
  down over = if move-down
  then then then then drop
; ok
Ok

\ Move each segment of the snake forward by one ok
: move-snake-tail 0 length @
  do
    i snake-x @ i 1 + snake-x !
    i snake-y @ i 1 + snake-y !
    -1 +loop ; ok
Ok

: is-horizontal direction @ dup
  left = swap
  right = or ; ok
ok

: is-vertical direction @ dup
  up = swap
  down = or ; ok
ok

: turn-up    is-horizontal if up    direction ! then ; ok
: turn-left  is-vertical   if left  direction ! then ; ok
: turn-down  is-horizontal if down  direction ! then ; ok
: turn-right is-vertical   if right direction ! then ; ok
Ok

: change-direction ( key -- )
  37 over = if turn-left else
  38 over = if turn-up else
  39 over = if turn-right else
  40 over = if turn-down
  then then then then drop ; ok
```

```
ok

: check-input
last-key @ change-direction
0 last-key ! ; ok
Ok

\ get random x or y position within playable area ok
: random-position ( -- pos )
width 4 - random 2 + ; ok
ok

: move-apple
apple-x @ apple-y @ draw-white
random-position random-position
set-apple-position ; ok
ok

: grow-snake 1 length +! ; ok
Ok

: check-apple
  snake-x-head @ apple-x @ =
  snake-y-head @ apple-y @ =
  and if
  move-apple
  grow-snake
  then
; ok
Ok

: check-collision ( -- flag )
  \ get current x/y position
  snake-x-head @ snake-y-head @
  \ get color at current position
  convert-x-y graphics + @
  \ leave boolean flag on stack
  0 =
; ok
Ok

: draw-snake
  length @ 0 do
  i snake-x @ i snake-y @ draw-black
  loop
  length @ snake-x @
  length @ snake-y @
```

```
  draw-white
; ok
Ok

: draw-apple
  apple-x @ apple-y @ draw-black ; ok
ok

ok
: game-loop ( -- )
  begin
    draw-snake
    draw-apple
    100 sleep
    check-input
    move-snake-tail
    move-snake-head
    check-apple
    check-collision
  until
  ." Game Over" ; ok
Ok

: start initialize game-loop ; ok
Ok
```



Before we delve too deeply into this code, two disclaimers. First, this is terrible Forth code. I'm by no means a Forth expert, so there's probably all kinds of things I'm doing in completely the wrong way. Second, this game uses a few non-standard techniques in order to interface with JavaScript. I'll go through these now.

Non-Standard Additions

The Canvas

You may have noticed that this editor is different from the others: it has an HTML5 Canvas element built in. I've created a very simple memory-mapped interface for drawing onto this canvas. The canvas is split up into 24 x 24 "pixels" which can be black or white. The first pixel is found at the memory address given by the variable `graphics`, and the rest of the pixels are offsets from the variable. So, for example, to draw a white pixel in the top-left corner you could run

```
1 graphics !
```

<- Top

The game uses the following words to draw to the canvas:

```
: convert-x-y ( x y -- offset )  24 cells * + ;    ( 101 )
: draw ( color x y -- )  convert-x-y graphics + ! ;
: draw-white ( x y -- )  1 rot rot draw ;          ( 102 )
: draw-black ( x y -- )  0 rot rot draw ;          ( 103 )
```

For example, `3 4 draw-white` draws a white pixel at the coordinates (3, 4). The y coordinate is multiplied by 24 to get the row, then the x coordinated is added to get the column.

Non-Blocking Keyboard Input

The Forth word `key` blocks, so is unsuitable for a game like this. I've added a variable called `last-key` which always holds the value of the last key to be pressed. `last-key` is only updated while the interpreter is running Forth code.

Random Number Generation

The Forth standard doesn't define a way of generating random numbers, so I've added a word called `random ( range -- n )` that takes a range and returns a random number from 0 to range - 1. For example, `3 random` could return `0`, `1`, or `2`.

sleep (ms --) (35)

Finally, I've added a blocking `sleep` word that pauses execution for the number of milliseconds given.

The Game Code

Now we can work through the code from start to finish.

Variables and Constants

The start of the code just sets up some variables and constants:

(**variables** - snake-x-head, snake-y-head – apple-x, apple-y – direction, length -)

(**constants** for - left, up, right, down – field width and height -)

```
variable snake-x-head
500 cells allot

variable snake-y-head
500 cells allot

variable apple-x
variable apple-y

0 constant left
1 constant up
2 constant right
3 constant down

24 constant width
24 constant height

variable direction
variable length
```

`snake-x-head` and `snake-y-head` are memory locations used to store the x and y coordinates of the head of the snake. 500 cells of memory are alloted after these two locations to store the coordinates of the tail of the snake.

Next we define two words for accessing memory locations representing the body of the snake.

```
: snake-x ( offset -- address )
  cells snake-x-head + ;

: snake-y ( offset -- address )
  cells snake-y-head + ;
```

Just like the `number` word earlier, these two words are used to access elements in the arrays of snake segments. After this come some words for drawing to the canvas, described above.

We use constants to refer to the four directions (`left`, `up`, `right`, and `down`), and a variable `direction` to store the current direction.

Initialization

After this we initialize everything:

```
: draw-walls
  width 0 do
    i 0 draw-black
    i height 1 - draw-black
  loop
  height 0 do
    0 i draw-black
    width 1 - i draw-black
  loop ;

: initialize-snake
  4 length !
  length @ 1 + 0 do
    12 i - i snake-x !
    12 i snake-y !
  loop
  right direction ! ;
```

```
: set-apple-position apple-x ! apple-y ! ;

: initialize-apple  4 4 set-apple-position ;

: initialize
  width 0 do
    height 0 do
      j i draw-white
    loop
  loop
  draw-walls
  initialize-snake
  initialize-apple ;
```

`draw-walls` uses two `do/loop`s to draw the horizontal and vertical walls, respectively.

`initialize-snake` sets the `length` variable to `4`, then loops from `0` to `length + 1` filling in the starting snake positions. The snake positions are always kept one longer than the length so we can grow the snake easily.

`set-apple-position` and `initialize-apple` set the initial position of the apple to (4,4).

Finally, `initialize` fills everything in white and calls the three initialization words.

Moving the Snake

Here's the code for moving the snake based on the current value of `direction`:

```
: move-up   -1 snake-y-head +! ;
: move-left   -1 snake-x-head +! ;
: move-down   1 snake-y-head +! ;
: move-right   1 snake-x-head +! ;

: move-snake-head  direction @
  left over  = if move-left else
  up over    = if move-up else
  right over = if move-right else
```

```
  down over  = if move-down
  then then then then drop ;

\ Move each segment of the snake forward by one
: move-snake-tail  0 length @ do
    i snake-x @ i 1 + snake-x !
    i snake-y @ i 1 + snake-y !
  -1 +loop ;
```

`move-up`, `move-left`, `move-down`, and `move-right` just add or subtract one from the x or y coordinate of the snake head. `move-snake-head` inspects the value of `direction` and calls the appropriate `move-*` word. This `over = if` pattern is an idiomatic way of doing case statements in Forth.

`move-snake-tail` goes through the array of snake positions backwards, copying each value forward by 1 cell. This is called before we move the snake head, to move each segment of the snake forward one space. It uses a `do/+loop`, a variation of a `do/loop` that pops the stack on every iteration and adds that value to the next index, instead of incrementing by 1 each time. So `0 length @ do -1 +loop` loops from `length` to `0` in increments of `-1`.

Keyboard Input

The next section of code takes the keyboard input and changes the snake direction if appropriate.

```
: is-horizontal  direction @ dup
  left = swap
  right = or ;

: is-vertical  direction @ dup
  up = swap
  down = or ;

: turn-up     is-horizontal if up direction ! then ;
: turn-left   is-vertical if left direction ! then ;
: turn-down   is-horizontal if down direction ! then ;
: turn-right  is-vertical if right direction ! then ;

: change-direction ( key -- )
```

```
  37 over = if turn-left else
  38 over = if turn-up else
  39 over = if turn-right else
  40 over = if turn-down
  then then then then drop ;

: check-input
  last-key @ change-direction
  0 last-key ! ;
```

`is-horizontal` and `is-vertical` check the current status of the `direction` variable to see if it's a horizontal or vertical direction.

The `turn-*` words are used to set a new direction, but use `is-horizontal` and `is-vertical` to check the current direction first to see if the new direction is valid. For example, if the snake is moving horizontally, setting a new direction of `left` or `right` doesn't make sense.

`change-direction` takes a key and calls the appropriate `turn-*` word if the key was one of the arrow keys. `check-input` does the work of getting the last key from the `last-key` pseudo-variable, calling `change-direction`, then setting `last-key` to 0 to indicate that the most recent keypress has been dealt with.

The Apple

The next code is used for checking to see if the apple has been eaten, and if so, moving it to a new (random) location. Also, if the apple has been eaten we grow the snake.

```
\ get random x or y position within playable area
: random-position ( -- pos )
  width 4 - random 2 + ;

: move-apple
  apple-x @ apple-y @ draw-white
  random-position random-position
  set-apple-position ;

: grow-snake  1 length +! ;
```

```
: check-apple ( -- flag )
  snake-x-head @ apple-x @ =
  snake-y-head @ apple-y @ =
  and if
    move-apple
    grow-snake
  then ;
```

`random-position` generates a random x or y coordinate in the range of `2` to `width - 2`. This prevents the apple from ever appearing right next to the wall.

`move-apple` erases the current apple (using `draw-white`) then creates a new pair of x/y coordinates for the apple using `random-position` twice. Finally, it calls `set-apple-position` to move the apple to the new coordinates.

`grow-snake` simply adds one to the `length` variable.

`check-apple` compares the x/y coordinates of the apple and the snake head to see if they're the same (using `=` twice and `and` to combine the two booleans). If the coordinates are the same, we call `move-apple` to move the apple to a new position and `grow-snake` to make the snake 1 segment longer.

Collision Detection

Next we see if the snake has collided with the walls or itself.

```
: check-collision ( -- flag )
  \ get current x/y position
  snake-x-head @ snake-y-head @

  \ get color at current position
  convert-x-y graphics + @

  \ leave boolean flag on stack
  0 = ;
```

`check-collision` checks to see if the new snake head position is already black (this word is called *after* updating the snake's position but

before drawing it at the new position). We leave a boolean on the stack to say whether a collision has occured or not.

Drawing the Snake and Apple

The next two words are responsible for drawing the snake and apple.

```
: draw-snake
  length @ 0 do
    i snake-x @ i snake-y @ draw-black
  loop
  length @ snake-x @
  length @ snake-y @
  draw-white ;

: draw-apple
  apple-x @ apple-y @ draw-black ;
```

`draw-snake` loops through each cell in the snake arrays, drawing a black pixel for each one. After that it draws a white pixel at an offset of `length`. The last part of the tail is at `length - 1` into the array so `length` holds the previous last tail segment.

`draw-apple` simply draws a black pixel at the apple's current location.

The Game Loop

The game loop constantly loops until a collision occurs, calling each of the words defined above in turn.

```
: game-loop ( -- )
  begin
    draw-snake
    draw-apple
    100 sleep
    check-input
    move-snake-tail
    move-snake-head
    check-apple
    check-collision
  until
  ." Game Over" ;
```

```
: start  initialize game-loop ;
```

The `begin/until` loop uses the boolean returned by `check-collision` to see whether to continue looping or to exit the loop. When the loop is exited the string `"Game Over"` is printed. We use `100 sleep` to pause for 100 ms every iteration, making the game run at rougly 10 fps.

`start` just calls `initialize` to reset everything, then kicks off `game-loop`. Because all the initialization happens in the `initialize` word, you can call `start` again after game over.

And that's it! Hopefully all the code in the game made sense. If not, you can try running individual words to see their effect on the stack and/or on the variables.

The End

Forth is actually much more powerful than what I've taught here (and what I implemented in my interpreter). A true Forth system allows you to modify how the compiler works and create new defining words, allowing you to completely customize your environment and create your own languages within Forth.

A great resource for learning the full power of Forth is the short book "Starting Forth" by Leo Brodie. It's available for free online and teaches you all the fun stuff I left out. It also has a good set of exercises for you to test out your knowledge. You'll need to download a copy of SwiftForth to run the code though.

####

Easy Forth Words - sort by page where found first

No	Page	Word	Explanation
1	2	+	Add the two top stack items – leave result on stack
2	3	*	
3	3	:	Start a new word : NEW xx yy zz ;
4	4	;	End the new word definition with a ;
5	4	dup	
6	5	drop	
7	5	swap	
8	5	over	
9	6	rot	
10	6	.	
11	6	emit	
12	7	cr	
13	7	." "	
14	8	Booleans - Flag	
15	8	=	
16	8	<	
17	8	>	
18	9	and	
19	9	or	
20	9	invert	
21	9	if	
22	10	else	
23	9	then	
24	10	do	
25	10	loop	
26	12	variable	
27	13	constant	
28	14	cells	

29	14	allot	
30	14	!	
31	14	?	
32	15	key	
33	15	begin	
34	15	until	
35	19	sleep	
101	19	convert x y	
102	19	draw white x,y	
103	19	draw black x,y	

Add other words that are not really explained :

( ) >R R>

Easy Forth Words - sort alphabetically

No	Page	Word	Explanation
30	14	!	
02	3	*	
10	6	.	
13	7	." ____ "	
03	3	:	
04	4	;	
31	14	?	
01	2	+	
16	8	<	
15	8	=	
17	8	>	
29	14	allot	
18	9	and	
33	15	begin	
14	8	Booleans - Flag	
28	14	cells	
27	13	constant	
12	7	cr	
24	10	do	
06	5	drop	
05	4	dup	
22	10	else	
11	6	emit	
21	9	if	
20	9	invert	
32	15	key	
25	10	loop	
19	9	or	
08	5	over	

09	**6**	**rot**	
35	**19**	**sleep**	
07	**5**	**swap**	
23	**9**	**then**	
34	**15**	**until**	
26	**12**	**variable**	
101	**19**	**convert_x_y**	
102	**19**	**draw_white x,y**	
103	**19**	**draw_black x,y**	

Snake Game 24 x 24 play area

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								
21																								
22																								
23																								
24																								
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16								

Notes: