not write a new program for this, simply extend your existing parser. Output is via SDL. You may find the function call `SDL_RenderDrawLine` useful.

Show a testing strategy on the above - you should give details of unit testing, white/black-box testing done on your code. Describe any test-harnesses used. In addition, give examples of the output of many different turtle programs. Convince me that every line of your C code has been tested.

Show an extension to the project in a direction of your choice. It should demonstrate your **understanding** of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar.

Hints

- All four sections above are equally weighted.
- Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

```
<MAIN>         ::= "{" <INSTRCTLST>
<INSTRCTLST>   ::= <INSTRUCTION><INSTRCTLST> |
                   "}"
<INSTRUCTION> ::= <FD> | <LT> | <RT>
<FD>           ::= "FD" <VARNUM>
<LT>           ::= "LT" <VARNUM>
<RT>           ::= "RT" <VARNUM>
<VARNUM>       ::= number
```

- The language is simply a sequence of words (even the semi-colons), so use `fscanf()`.
- Some issues, such as what happens if you use an undefined variable, or if you use a variable before it is set, are not explained by the formal grammar. Use your own common-sense, and explain what you have done.
- Once your parser works, extend it to become an interpreter. DO NOT aim to parse the program first and then interpret it separately. Interpreting and parsing are inseparably bound together.
- Start testing very early - this is a complex beast to test and trying to do it near the end won't work.

Submission

Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the parser, interpreter and extension sections, make sure there's a `Makefile`, so that I can easily build the code using `make parse`, `make interp` and `make extension`. Submit a single `turtle.zip` file.

■

12.4 NLab

- The programming language MATLAB (originally available in the late 1970s, for free) is one of the most widely used scientific langaues in the world.
- One of the most interesting things about MATLAB, is that every single variable is stored as a $2D$ array - even a scaler integer is simply a 1×1 array [1].
- Here, we develop a very simple version of this concept - a language that allows such arrays to be created or read from file, and functions performed on each part of the array, one

[1] Actually as the name implies, they are all stored as matrices, but we will ignore the mathematical interprettion here.

element at a time.

Examples

```
BEGIN {
   SET $I := 5 ;
   PRINT $I
}
```

sets the variable I to have the value 5, and prints it to the screen:

```
5
```

You can create an array full of ones and add 2 to each cell of the array:

```
BEGIN {
   ONES 6 5 $A
   SET $A := $A 2 B-ADD ;
   PRINT
   PRINT $A
}
```

```
ARRAY:
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
```

Loops are possible too, here a loop counts from 1 to 10 via the variable I and computes factorials in the variable F. Both variables are scalars (a 1×1 array) :

```
BEGIN {
   SET $F := 1 ;
   LOOP $I 10 {
      SET $F := $F $I B-TIMES ;
      PRINT $F
   }
}
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

Such loops (like in C) have counters stored in a variable. Changing this variable inside the loop can affect when the loop ends :

```
# Notice that the loop counter is modified inside the loop
# causing it to count at twice the speed : 2 4 6 8 10
BEGIN {
   LOOP $I 10 {
      SET $I := $I 1 B-ADD ; PRINT $I
   }
```

```
}
```

```
2
4
6
8
10
```

As grammar tells you, loops can be nested too :

```
BEGIN {
    SET $A := 0 ;
    LOOP $I 5 {
        LOOP $J 5 {
            SET $A := $I $J B-TIMES ;
            PRINT $A
        }
    }
}
```

```
1
2
3
4
5
2
4
6
8
10
3
6
9
12
15
4
8
12
16
20
5
10
15
20
25
```

The Formal Grammar

```
<PROG> ::= BEGIN "{" <INSTRCLIST>

<INSTRCLIST> ::= "}" | <INSTRC> <INSTRCLIST>
<INSTRC> ::= <PRINT> | <SET> | <CREATE> | <LOOP>

# Print array or one-word string to stdout
<PRINT> ::= "PRINT" <VARNAME> | "PRINT" <STRING>

# One of the 26 possible (upper-case) variables
<VARNAME> ::= $[A-Z] % e.g. $A, $B, $Z etc.

# Because of the assumption that a program is just a list of words,
strings can't have spaces in them (for simplicity)
<STRING> :: Double-quoted string e.g. "../../doof.arr", "Hello!" etc.

<SET> ::= <VARNAME> ":=" <POLISHLIST>
<POLISHLIST> ::= <POLISH><POLISHLIST> | ";"
```

```
<POLISH> ::= <PUSHDOWN> | <UNARYOP> | <BINARYOP>
<PUSHDOWN> ::== <VARNAME> | <INTEGER>

# A non-negative integer
<INTEGER> ::= [0-9]+ % e.g. 1, 250, 3

# Pop one array, push the result.
# U-NOT : Flip the Boolean values of an array
# U-EIGHTCOUNT : Returns the numbers of true values around each cell in the array in its
# Moore 8-neighbourhood (north, south, west, east, NE, NW, SW, SE).
<UNARYOP> ::= "U-NOT" | "U-EIGHTCOUNT"

# Pop 2 arrays, push the resultant array
# If both arrays are bigger than 1x1, they must be the same size
# If one array is a 1x1 scalar, apply this value to each cell of the other array in turn
# B-TIMES operates on corresponding cells in turn (it is not a full matrix multiplication).
<BINARYOP> :: "B-AND" | "B-OR" | "B-GREATER" | "B-LESS" | "B-ADD" | "B-TIMES" | "B-EQUAL"

# Create an array full of ones, or read from a file
<CREATE> ::= "ONES" <ROWS> <COLS> <VARNAME> | "READ" <FILENAME> <VARNAME>
<ROW> ::= <INTEGER>
<COL> ::= <INTEGER>
<FILENAME> ::= <STRING>

# Loop using a variable to count from 1 (!) to <= <INTEGER>
# IF the variable
<LOOP> ::= "LOOP" <VARNAME> <INTEGER> "{" <INSTRCLIST>
```

Exercise 12.4

- 30% Implement a recursive descent parser - this will report whether or not a given NLab program follows the formal grammar or not. The input file is specified via `argv[1]` - there is **no** output if the input file is **valid**. Elsewise, a non-zero `exit` is made.
- 30% Extend the parser, so it becomes an interpreter. The instructions are now 'executed'. Do not write a new program for this, simply extend your existing parser.
- 20% Show a testing strategy on the above - you should give details of unit testing, white/black-box testing done on your code. Describe any test-harnesses used. In addition, give examples of the output of many different NLab programs. Convince me that every line of your C code has been tested.
- 20% Show an extension to the project in a direction of your choice. It should demonstrate your **understanding** of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar.

Hints

Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

- ```
  <PROG> ::== "BEGIN" { <INSTRCLIST>
  INSTRCLIST ::= "}" | <INSTR> <INSTRCLIST>
  <INSTR> ::= <PRINT> | <SET>
  <PRINT} ::= "PRINT" <VARNAME>
  <SET> ::= <VARNAME> ":=" <POLISHLIST>
  <POLISHLIST> ::= <POLISH> <POLISHLIST> | ";"
  <POLISH> ::= <VARNAME> | <INTEGER>
  ```

- The language is simply a sequence of words (even the semi-colons), so use `fscanf()`.
- Some issues, such as what happens if you use an undefined variable, or if you use
  ```