```
{
   FOR $I = $@ TO 1 STEP -1 {
      PRINT $[ $I ]
   }
   PRINT "\n"
}
```

Exercise 12.4 Write a C program to implement the above formal grammar. Your program should read in a cawk program (argv[1]) and expect the data file to be read from standard input (or from argv[2] if specified).

The marks are split as follows:

- (25%) To implement a recursive descent parser - this says whether or not a given CAWK program follows the formal grammar or not.
- (25%) To implement an interpreter, so that the instructions are executed.
- (25%) To show a testing strategy on the above - you should give details of white-box and black-box testing done on your code. Describe any test-harnesses used. Give examples of the output of many different cawk programs.
- (25%) To 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.

Submit the program(s) and a Makefile so that I can:

- Compile the parser by typing 'make parse'.
- Compile the interpreter by typing 'make interp'.
- Compile the extension by typing 'make extension'.

In addition:

- Submit a test strategy report called test.txt. This will include sample outputs, any code written especially for testing etc.
- Submit an extension report called 'extend.txt'. This is quite brief and explains the extension attempted.

■

12.5 Neill's Adventure Language

Some of the very earliest computer games were text-based adventures e.g. Colossal Cave

https://en.wikipedia.org/wiki/Colossal_Cave_Adventure

Here we write a simple language (NAL) which allows us to write (simplified) versions of such games, focussing on setting variables and printing. The grammar is as follows:

```
<PROGRAM> := "{" <INSTRS>
<INSTRS> := "}" | <INSTRUCT> <INSTRS>
<INSTRUCT> := <FILE> | <ABORT> | <INPUT> | <IFCOND> | <INC> | <SET> |
                  <JUMP> | <PRINT> | <RND>

% Execute the instructions in file, then return here e.g. :
% FILE "test1.nal"
<FILE> := "FILE" <STRCON>

% Halt/abort all execution right now !
<ABORT> := "ABORT"
```

```
% Fill a number−variable with a number, or 2 string−variables with string :
% IN2STR ( $C, $ZER ) or INNUM ( %NV )
<INPUT> := "IN2STR" "(" <STRVAR> "," <STRVAR> ")" | "INNUM" "(" <NUMVAR> ")"

% Jump to the nth word in this file (the first word is number zero!)
% Brackets count as one word, "things in quotes" count as one word, e.g. :
% JUMP 5
<JUMP> := "JUMP" <NUMCON>

% Output the value of variable, or constant, to screen with (without a linefeed)
<PRINT> := "PRINT" <VARCON>
<PRINTN> := "PRINTN" <VARCON>

% Set a variable to a random number in the range 0 − 99 e.g. :
% RND ( %N )
% Number should be seeded via the clock to be different on successive executions
<RND> := "RND" "(" <NUMVAR> ")"

% If condition/test is true, execute INSTRS after brace, else skip braces
<IFCOND> := <IFEQUAL> "{" <INSTRS> | <IFGREATER> "{" <INSTRS>
<IFEQUAL> := "IFEQUAL" "(" <VARCON> "," <VARCON> ")"
<IFGREATER> := "IFGREATER" "(" <VARCON> "," <VARCON> ")"

% Add 1 to a number−variable e.g. :
% INC ( %ABC )
<INC> := "INC" "(" <NUMVAR> ")"

% Set a variable. All variables are GLOBAL, and persist across the use of FILE etc.
% $A = "Hello" or %B = 17.6
<SET> := <VAR> "=" <VARCON>

% Some helpful variable/constant rules
% (Here ROT18 is ROT13 for letters and rot5 for digits)
<VARCON> := <VAR> | <CON>
<VAR> := <STRVAR> | <NUMVAR>
<CON> := <STRCON> | <NUMCON>
<STRVAR> := $[A−Z]+
<NUMVAR> := %[A−Z]+
<STRCON> := A plain−text string in double−quotes, e.g. "HELLO.TXT",
                or a ROT18 string in hashes e.g. #URYYB.GKG#
<NUMCON> := A number e.g. 14.301
```

Note that string constants can be entered via the use of double quotes, or using hashes to encode strings using ROT18

`https://en.wikipedia.org/wiki/ROT13`

in which characters are encoded/decoded according to:

```
Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ
ROT13: NOPQRSTUVWXYZABCDEFGHIJKLM
Plain: abcdefghijklmnopqrstuvwxyz
ROT13: nopqrstuvwxyzabcdefghijklm
Plain: 0123456789
ROT5 : 5678901234
```

the algorithm allows obvious ‘spoilers’ to be hidden from users browsing through `.nal` files, but is simple to apply.

A simple program showing the use of assignment, conditionals and ROT18 is shown:

```
{
    $A = "Neill"
    %E = 12.4
    IFEQUAL ( $A , #Arvyy# ) {
        PRINT #Uryyb Jbeyq!#
    }
}
```

Here string-variables (`$`) and number-variables (`#`) are initialised.

The use of ‘JUMP’ is shown next - this allows execution to move a chosen word in the program. Strings inside quotes count as one word, so the following contains six words:

```
{
    PRINT "Warning : Infinite Loop !"
    JUMP 1
}
```

‘FILE’ allows another file to be executed (as <PROG>) and then execution returns to the original when finished. All variables are shared across files and are global:

```
{
    PRINT "In test4, before"
    FILE "test1.nal"
    PRINT "In test4, after"
}
```

‘RND’ sets a variable to a number in the range $0-99$, while ‘INC’ adds one to a number-variable.

```
{
    %C = 0
    RND ( %A )
    PRINT %A
    INC ( %C )
    IFGREATER ( %C , 9 ) {
        ABORT
    }
    JUMP 4
}
```

‘ABORT’ halts execution instantly.

A simple guessing game is shown below:

```
{

    PRINT "I'm thinking of a number (0-99).\nCan you guess it?"
    RND ( %MINE )
    %CNT = 0

    INC ( %CNT )
    PRINT "Type in your guess"
    INNUM ( %GUESS )
    IFGREATER ( %CNT , 7 ) {
        PRINT #Gbb znal gevrf :-(#
        ABORT
```

```
    }
    IFGREATER ( %GUESS , %MINE ) {
        PRINT "Too Big ! Try again ... "
        JUMP 10
    }
    IFGREATER ( %MINE , %GUESS ) {
        PRINT "Too Small ! Try again ... "
        JUMP 10
    }
    IFEQUAL ( %MINE , %GUESS ) {
        PRINT #Lbh thrffrq pbeerpgyl, lbh jva :-)#
        PRINTN "Number of goes = "
        PRINT %CNT
        ABORT
    }

}
```

'INNUM' gets a number (float) from the user, and the use of 'PRINT' (with a newline after), and 'PRINTN' (without a newline after) is demonstrated.

Exercise 12.5

- **(40%)** Implement a parser. The `.nal` file should be read in using `argv[1]`. If the file is parsed correctly, the only out should be:

  ```
  Parsed OK
  ```

- **(30%)** Implement an interpreter, building on top of the parser in the manner described in the lectures. Do not write a brand new program - interpretation will be done alongside parsing.
- **(20%)** Submit the testing you have undertaken, including examples and a description of your strategies. This should convince us that you have tested every line of code, and that it works correctly. If there are still issues/bugs state them clearly. Also, point out any bugs that you have successfully found using these approaches. Submit a file named `testing.txt`, along with any other files you feel necessary. Due to the recursive nature of this assignment testing is non-trivial - simply submitting many `.nal` files that 'work' is not sufficient. No particular strategy is mandated. You may wish to explore a couple and briefly discuss strengths and weaknesses.
- **(10%)** Undertake an extension of your choosing. Remember, that the assessment is based on the quality of your coding, so choose something to demonstrate an aspect of programming or software engineering that you haven't had a chance to use in the main assignment. Submit a file named `extension.txt` outlining, in brief, your contribution.

Hints

Don't try to write the entire program in one go. Try a cut down version of the grammar first. Build-up from the `01s` example given in lectures.

- 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.

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 one `Makefile`, so that I can easily build the code using `make parse`, `make interp` and `make extension`. I've given an example `makefile` in the usual place, but this is an example only - yours may be different. I wrote only one program `nal.c` and built the two different version by setting a `#define` **via the makefile with** `-DINTERP`. Inside the code itself `#ifdef INTERP` and `#endif` are used. Also make sure that basic testing is available using `make testparse` and `make testinterp`.
Place all the files required for your submission in a single `.zip` file called `nal.zip` - this file will not contain other zipped files.

■