12.7 Turtle Graphics

History

Many attempts have been made to create programming languages which are intuitive and easy to learn. One of the best of these was *LOGO* which allowed children as young as 3 to learn a computer language. A subset of this language involved a "turtle" which could be driven around the screen using simple instructions.

An Example

At its most basic, the turtle is driven by simple instructions such as `FORWARD` to move the turtle and leave a trail behind it, and `RIGHT` which rotates the turtle clockwise :

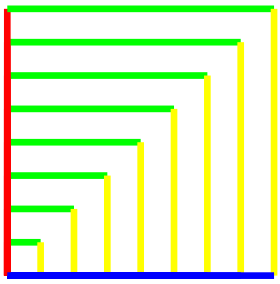
```
START
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
END
```

This may be simplified by using a loop to repeat an operation several times. Here our grammar allows a variable to iterate over each item in a list :

```
START
  LOOP A OVER { 1 2 3 4 5 6 7 8 }
    FORWARD 5
    RIGHT 45
  END
END
```

Here the actual values of the variable `A` weren't really important since the variable wasn't used directly inside the loop itself. Colours can be set by using string constants such as `"RED"` or `"CYAN"` :

```
START
  LOOP D OVER { 1 2 3 4 5 6 7 8 }
    LOOP C OVER { "RED" "GREEN" "YELLOW" "BLUE" }
      COLOUR $C
      FORWARD $D
      RIGHT 90
    END
  END
END
```



Variables can be set using a postfix notation, e.g. :

```
START
  SET A ( 0 )
  SET B ( $A 1 + )
  SET C ( $B 2 * )
END
```

The Formal Grammar

```
<PROG>   ::= "START" <INSLST>

<INSLST> ::= "END" | <INS> <INSLST>
<INS>    ::= <FWD> | <RGT> | <COL> | <LOOP> | <SET>

<FWD>    ::= "FORWARD" <VARNUM>
<RGT>    ::= "RIGHT" <VARNUM>
<COL>    ::= "COLOUR" <VAR> | "COLOUR" <WORD>
<LOOP>   ::= "LOOP" <LTR> "OVER" <LST> <INSLST>
<SET>    ::= "SET" <LTR> "(" <PFIX>

<VARNUM> ::= <VAR> | <NUM>
% Variables e.g. $A, $B, $Z etc.
<VAR>    ::= $<LTR>
% One Uppercase letter
<LTR>    ::= A, B ... Z
% Any valid double (as defined by scanf("%lf"...)
<NUM>    ::= 10 or -17.99 etc.

% A single word (as defined by scanf("%s"...) with double-quotes around it
% Valid colours include "BLACK", "RED", "GREEN", "BLUE",
% "YELLOW", "CYAN", "MAGENTA", "WHITE"
<WORD>   ::=  "RED", "BLUE", "HELLO!" or  "178"

<LST>    ::= "{" <ITEMS>
<ITEMS>  ::= "}" | <ITEM> <ITEMS>
<ITEM>   ::= <VARNUM> | <WORD>

<PFIX>   ::= ")" | <OP> <PFIX> | <VARNUM> <PFIX>
% A single mathematical operation character
<OP>     ::= + - / *
```

Exercise 12.7.1

Parser (35%)

Implement a recursive descent parser - this will report whether or not a given turtle 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 graceful non-zero `exit` is made. You should use the techniques described in the lecture for this - writing code that carefully follows the grammar. For instance, it won't use a tokenzier.
All source code will be in the `Parse/` sub-directory.

Interpreter (25%)

Extend the parser, so it becomes an interpreter. The instructions are now 'executed'. Do not begin a new program for this, simply copy and then extend your existing parser. Output is to a text file if the users specifies an `argv[2]`, in the form of a 2D array of characters where colours are represented by blac(K), (R)ed, (G)reen, (Y)ellow, (B)lue, (M)agenta, (C)yan and

(W)hite, via a 51 wide × 33 height array of characters e.g.:

```
   GGGGGY
   R    Y
   R    Y
   R    Y
   R    Y
WWWWWWWWWWWW
 W         W
 W         W
  W       W
  W       W
   W     W
   W   W
    W  W
    W W
     WW
     W
```

If no `argv[2]` is specified, then output is to the screen using ANSI colour characters in same style as that used in Section 5.2, with a 1 second pause after each `FORWARD` instruction.
Note that some files that successfully parse will not successfully interpret (and vice versa) - there are example `.ttl` files provided to demonstrate this.
All interpreter source code will be in the `Interp/` sub-directory.

Postscript (5%)
Extend the output functionality of the interpreter such that if `argv[2]` has a `.ps` extension, then the correct postscript instructions are written to that file.

www `https://paulbourke.net/dataformats/postscript`

Since PDFs are a more common file format, then at the end, your program should also convert the `.ps` file to a `.pdf` file using a `system()` call in your C code to `ps2pdf` which is installed on the lab machines. My implementation uses the postscript commands `showpage`, `newpath`, `moveto`, `lineto`, `setrgbcolour`, `scale` and `stroke`.

If you get this to work, add it to the interpreter above - it will be tested using both `.txt` and `.ps` output modes.

Extension (15%)

Extend 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 give us the new, full grammar, along with the code in the `Extension/` sub-directory named `grammar.txt`, in addition to a 300 word (max) description of what you've achieve in `extension.txt`. Please do not submit ideas for future work, only functionality you've got working.

Testing (20%)

Show the testing strategy on your code - you should give details of any unit testing or white/black-box testing done on your code. Describe any test-harnesses used. Convince us that every line of your C code has been tested, explaining the process, lessons learned and bugs found. Submit a file `testing.txt` in the sub-directory `Testing/`.

Hints

- Understand the noughts and ones example given in the notes.
- Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

```
<PROG>   ::= "START" <INSLST>

<INSLST> ::= "END" | <INS> <INSLST>
<INS>    ::= <FWD> | <RGT>

<FWD>    ::= "FORWARD" <NUM>
<RGT>    ::= "RIGHT" <NUM>

<NUM>    ::= 10 or -17.99 etc.
```

- The language is simply a sequence of words (even the braces), so use `fscanf()`.
- Some issues, (e.g. drawing out-of-bounds), incorrect postfix expressions cannot explained by the formal grammar. Use your own common-sense to decide how to reconcile these, 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. Use our lectures notes for this and not those from other units such as Architecture which take a very slightly different approach.
- For the simple trigonometry required to compute the turtles destination, bear in mind that the `cos()` and `sin()` functions in C require parameters in radians and not degrees.
- The turtle always begins in the centre of the screen facing due north (upwards) and having a default white colour. For the postscript/PDF part, since white ink is hard to see on white paper, I've redefined white ink to be slightly grey.
- Start testing very early - this is a complex program to test and trying to do it (or explain it) near the end won't work.

Submission

Adapt the given Makefile if necessary to allow the parser, interpreter and extension to be built. Submit your extension via text files `extension.txt` in `Extension/` along with `grammar.txt` and your source code.

Your testing strategy will be explained in `testing.txt` along with any other files you'd like to bring our attention to in `Testing/`.