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 :

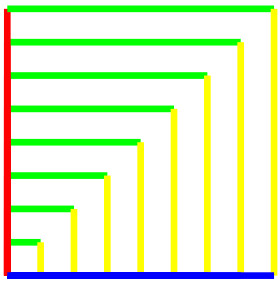
```
START
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
  FORWARD 5
  RIGHT 45
END
```

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/`.

Bundle this all up into a single `turtle.zip file` that contains **everything** required for us to compile and understand your code.

■

A. House Style

A.1 Correctness

These style rules ensure your code is as-correct-as-can-be with the aid of the compiler and other tools:

FLAGS Having no warnings (or errors!) when compiling and executing with the flags:

For array bounds checking, `NULL` pointers being dereferenced etc:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99
-fsanitize=undefined -fsanitize=address -g3
```

For memory leaks:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99
-g3
```

then run:

```
valgrind --leak-check=full ./myexec
```

For 'final' production-ready code:

```
-Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99
-O3
```

You can use more flags than this, obviously, but these will make sure a few of the essential warnings that commonly indicate the presence of bugs and leaks are checked. These guidelines are meant to be independent of the particular compiler used though. Sometimes it is helpful to use many compilers too, e.g. `gcc` and `clang`.

If you have unused variables (for example) in your code, it doesn't matter whether your compiler happened to tell you about it or not - it's still wrong !

BRACE Always brace all functions, `fors`, `whiles`, `if/else` etc. Somewhat contraversial, this ensures that 'extra' lines tagged onto loops are dealt with correctly. For instance:

```
while(i < 10)
    printf("%i\n", i);
    i++;
```

looks like it should print out `i` 10 times, but instead runs infinitely. The programmer probably meant:

```
while(i < 10){
    printf("%i\n", i);
    i++;
}
```

GOTO You do not use any of the keywords `continue`, `goto` or `break`. The one exception is inside `switch`, where `break` is allowed because it is essential ! These keywords usually lead to tangled and complex 'spaghetti' coding style. I often recommend that you rewrite the offending code uing functions, which **can** have multiple `returns` in them.

NAMES Meaningful identifiers. Make sure that functions names and variables having meaningful, but succinct, names.

REPC Repetitive code. If you've cut-and-paste large chunks of code, and made minor changes to it, you've done it wrong. Make it a function, and pass parameters that make the changes required.

```
int inbounds1(int i){
    if(i >=0 && i < MAX){
        return 1;
    }
    else{
        return 0;
    }
}

int inbounds2(int i){
    if(i >=0 && i < LEN){
        return 1;
    }
    else{
        return 0;
    }
}
```

might make more sense as:

```
int inbounds2(int i, int mx){
    if(i >=0 && i < mx){
        return 1;
    }
    else{
        return 0;
    }
}
```

GLOB No global variables. Global variables are declared 'above' `main()`, are in scope of all functions, and can be altered **anywhere** in the code. This makes it rather unclear **which** functions should be reading or writing them. You can make a case for saying that occasionally they could be useful (or better) than the alternatives, but for now, they are banned !

RETV Any functions that returns a value, should have it used:

```
scanf("%i", &i);
```

is incorrect. It returns a value that is ignored. Instead do:

```
if(scanf("%i", &i) != 1{
    /* PANIC */
```

The only exceptions are `printf` and `putchar` which do return values but which are typically ignored.

MATCH For every `fopen` there should be a matching `fclose`. For every `malloc` there should be a `free`. This helps avoid memory leaks, when your program or functions are later used in a larger project.

STDERR When exiting your program in an error state, make sure that you `fprintf` the error on `stderr` and not `stdout`. Use `exit`, e.g.

```
if(argc != 2){
    fprintf(stderr, "Usage : %s <filename>\n", argv[0]);
    exit(EXIT_FAILURE);
}
```

A.2 Prettifying

These rules are about making your code easier to read and having a consistent style in a form that others are expecting to see.

LLEN Line length. Many people use terminal and editors that are of a fixed-width. Having excessively long lines may cause the viewer to scroll to off the screen. Keep lines short, perhaps < 60 characters. However, in a similar way to the **FLEN** rule below, it's really about the complexity of the line that's the issue, not its absolute length. A programmer would generally find:

```
bool arrcleanse(cell oldarr[HEIGHT][WIDTH], cell newarr[HEIGHT][WIDTH], int h, int w)
```

a great deal easier to read than:

```
if(a < b && j++ >= szpar(e ? true : false) || h==4){
```

despite it being twice as long.

TABS Don't use tabs to indent your code. Every editor views these differently, so you have no guarantee that I'm seeing the same layout as you do. Use spaces. This also prevents issues when cutting-and-pasting from one source to another.

INDENT Indentation: choose a style for indentation and keep to it. I happen to use 3 spaces, put opening braces for functions on a new line, but at the end of `if`,`else`, `for`, `while` etc, then close them on a new line, underneath the 'i' of the `if`:

```
int smallest(int a, int b)
{
    if(a < b){
        return a;
    }
    else{
        return b;
    }
}
```

You can use any style you like, as long as it's consistent.

MAIN The code should have function prototypes/definitions first, then `main()`, followed by the function implementation. This means the reader always know where to find `main()`, since it will be near the top of the file.

CAPS Constants are `#defined`, and use all CAPITALS. For instance:

```
#define WEEKS 52
#define MAX(a,b) (a < b ? b:a)
```

FLEN Short functions. All functions are short. It's quite difficult to put a maximum number of lines on this, but use 20 as a starting point. Exceptions include a function that simply prints a list of instructions. There would be no benefit in splitting it into smaller functions. Short functions are easier to plan, write and test.
I find it more useful to think about how hard the function is to understand, rather than its length. Therefore, a 30 line, simple function is fine, but an extremely complex and dense 15 line function might need to be split up, or more self-documentation added.

A.3 Readability

Your code should be self-documenting. Comments will be written when there is something complex to explain, and only read when something has gone catastrophically wrong. In many cases clever use of coding will avoid the need for them. The compiler never sees them, so cannot check them. If you change your code, but not your comments, they can be highly misleading.

As Kevlin Henney said :

> A common fallacy is to assume authors of incomprehensible code will somehow be able to express themselves lucidly and clearly in comments.

MAGIC No magic numbers. There should be no inexplicable numbers in your code, such as:

```
if(i < 36){
```

It's probably unclear to the reader where the 36 has come from, or what it means, even if it is obvious to the programmer at the time of writing the code. Instead, `#define` them with a meaningful name. Array overruns are often cured by being consistent with `#defines`.

BRIEF Comments are brief, and non-trivial. Worthless commenting often looks something like:

```
// Set the variable i to zero
int i = 0;
```

The programmer extracts no additional information from it. However, for more difficult edge cases, a comment might be useful.

```
// Have we reached the end of the list ?
if(t1->h == NULL){
```

To prevent lines from becoming too long, it is good practice to put comments above the line it refers to, not at the end of the same line.

TYPE You should use `typedefs`, `enums` and `structs` to increase readability.

INFIN No loops should be infinite. I'll never ask you to write a program that is meant to run forever. Therefore statements such as

```
while(1){
```

or

```
for(;;){
```

are to be avoided.

2DINDEX 2D Arrays in C are indexed `[row][col]`. Sometimes it may still work correctly, especially if you've consistently confused the two. Therefore, if you write code that indexes it `[col][row]`, or `[x][y]` it will confuse anyone else trying to understand (or reuse) your code. If you were to sketch a graph using `(i,j)` you'd almost certainly make *i* the horizontal axis, and *j* the vertical. Therfore, for any two variables it makes more sense to write `[b][a]` or `[j][i]`.

B. Peer Assessment

B.1 Submitting

This year, you have the option of submitting one of your Chapter 2 exercises to be peer reviewed. This process is optional - there is no mark associated with this. However, we think it's good practice to learn how to submit your code through Blackboard, gain an insight into the English Universities marking system, and to think more deeply about how the marking criteria are applied.

To submit your work, go to the Unit Blackboard page, then *Assessment Submission and Feedback*, then *Peer Assessment*. You can submit as many times as you like - old files will be automatically archived.

Make sure your file has the correct name - this will be shown on the *Unit Information / Weekly Schedule* part of the Blackboard page.

B.2 Marking Criteria

When files are ready to be marked, we'll put them in the *Files* folder of the *General* channel of the unit 'Teams' group. We'll tell you which usernames you are assessing.

When you give feedback, you should use the following guidelines :

- If the file has been submitted late then I'll rename it to have *_late* in the name. In this case, take 10 off the final mark (a University rule).
- Our system will automagically prefix your username to the filename. Apart from this, if the file was misnamed in any way, even just using a `.C` rather than a `.c` extension, take 5 off the final mark.

Below, where a category has a fixed mark (e.g. choose 0 or 10), decide which of these two marks to award. Where a category has a range of marks (e.g. 0 – 20), apply the following guidelines:

50% Just OK - room for improvement.
60% Good, solid, work - what you might expect.
70% Above and beyond what we would expect at this stage iof the unit (just!)
80% We could publish this, ground-breaking - it'll change how we teach this unit in the future !

So, if a category can be marked in the range 0 – 20, a very good piece of work might score 13 marks, for instance.

Using the House Style rules, shown in Appendix A, give a mark and comment on each of the following items:

FLAGS - Does the code compile using clang and the house-style flags : -Wall -Wextra -Wfloat-equal -Wvla -pedantic -std=c99 -O2 with zero warnings?	**0 or 10**
GOTO + TABS + INDENT	**0 – 10**
BRACE	**0 – 5**
NAMES	**0 – 5**
LLEN	**0 – 5**
MAIN	**0 – 5**
CAPS	**0 – 5**
FLEN Including main()	**0 – 5**
MAGIC	**0 – 5**
BRIEF	**0 – 5**
TYPE	**0 – 5**
∗ Are all functions thoroughly tested using a function called test() or similar?	**0 – 20**
∗ Does the program work correctly?	**0 or 5**

Write some brief feedback about the program(s) you've been assigned and email it to those people using their usernames. An example is given below:

> This program compiled for me without warnings and was correctly named. The code was consistently indented, but you used tabs not spaces as required. Function names could have been improved upon - e.g. `doit()` is not very helpful. One function (`extra()`) seems quite long. You used a 'magic number' : 65. The testing was quite thorough and the program worked well. Overal I'd give this 59% - nicely done, but pay closer attention to the style guidelines!