

Assignment 2 - Graphical Modelling

Due Date: Friday 29th of March 23:59

Marks: 15 marks (plus up to 2 bonus marks)

Aims

This assignment aims to give you practice in:

- developing a solution to a problem
- developing a C program implementation of your solution
- writing a properly documented C program that adheres to the Style Guide
- Using and Manipulating Strings
- Using and Manipulating Linked Lists
- Using the math.h library
- Devising tests to test your solution
- Maintaining a blog of the work you do

You MUST use a linked list to store the start string. Failing to do so results in 0 marks for the assignment.

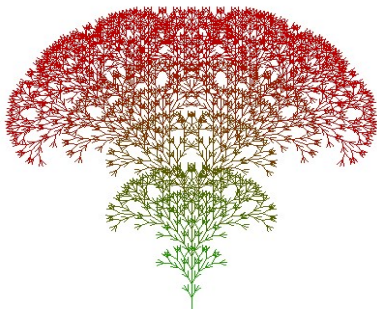
Contents

Introduction
Stage 0
Stage 1
Stage 2
Stage 3
Bonus Stage
Blog
Assessment
Submission

Introduction

Back in the '60s a biologist called Aristid Lindenmayer found that formal grammars can be used as a rather nice way of modelling the growth of colonies of bacteria. He went on to develop a theory of "L-Systems" which he used to model many different biological organisms, in particular their structure and growth patterns. He used his theory to generate images of a wide variety of plants, trees and flowers. Many of his pictures are incredibly realistic and yet they are generated from very concise descriptions.

The book "[The Algorithmic Beauty of Plants](#)" is available online and describes this topic in great detail. **You do not have to read it** but you might like to look at some of the pictures. Also the early chapters which describe the basics of L-Systems have some nice examples that you might like to try out when you have completed the last stage of the assignment.



Formal Grammars

For the purpose of this assignment, it is sufficient to view a formal grammar as a start string, together with a set of rules that specify how certain letters in the string are replaced by one or more new characters. For example, we have a start string

"F"

and a rule

```
'F' -> "F+F"
```

which means: replace the character 'F' with the string "F+F". So, applying the rule once to our start string, we get the new string

```
"F+F"
```

Now, we can apply the rule again, and get

```
"F+F+F+F"
```

and again

```
"F+F+F+F+F+F+F+F"
```

The string obviously grows very quickly, because every F gets replaced by a new string, which in turn contains more Fs.

Drawing Commands

Now, what is the connection between these grammars and the pictures? Well, we can define our own language where we interpret each character in the string as a drawing command. Since each rule application generates a bigger string, each application step can generate a more complicated picture.

The language we start with is very simple, and is similar to "Logo turtle graphics" style drawing commands. It supports the following 12 commands:

- F: draw a line from the current position a constant distance forward in the current direction.
- E: move from the current position a constant distance forward in the current direction without drawing.
- +: turn clockwise by a constant angle.
- -: turn anti-clockwise by a constant angle.
- R: increase the amount of red by a constant amount
- r: decrease the amount of red by a constant amount
- G: increase the amount of green by a constant amount
- g: decrease the amount of green by a constant amount
- B: increase the amount of blue by a constant amount
- b: decrease the amount of blue by a constant amount
- [: means save the current position, color and direction so we can come back to it later
-]: means return to the most recently saved position,color and direction

All other characters in the string should be ignored.

For example:

- The string "FfF" would be drawn as a straight line followed by a gap followed by another line.
- If we use a constant angle of 90 degrees for '+' then the string "F+F+F+F" would be drawn as a square.
- Again, using a constant angle of 90 degrees, the string "[F]+[F]+[F]+[F]" would be drawn as a cross. Remember that the "[...]" means that the position and angle is saved and restored.

The graphics output format is just a list of coloured lines in absolute coordinates. This is described in more detail in stage 1.

Input

Standard Input

When you run your program, it should read in the following from standard input:

- **Line length:** a floating point number that represents the constant distance that is used when generating line output data. You may assume that this will be greater than or equal to 0.
- **Initial Direction:** a floating point number that represents the initial direction in radians
- **Angle Increment:** a floating point number that represents the constant angle in radians that used to update the current direction when generating line output data
- **Initial Colour:** 3 floating point numbers between 0 and 1 inclusive that represent the initial colour in RGB (Red Green Blue) format.
- **Start String:** On the next line, we read in a string that represents the start string of the formal grammar.
- **Rules:** The rest of the input represents the rules of the grammar. Each rule is specified on two lines, the first line containing one character and the next containing a string. For example:

```
F
F+F
a
a++
```

Would represent 2 rules:

```
'F' -> "F+F"
```

and

```
'a' -> "a++"
```

You should continue reading in rules until a maximum of 20 rules have been read in or until we reach the end of file (ctrl -d).

Note: You may assume you will only need to read in strings (ie the start string and the strings that represent rules in the grammar) of at most 100 characters (not including the newline character or the end of string terminator characters).

Note: You can assume during automarking that the input will always be in the same format and will be valid. However you should try to make sure your program does something sensible when it is confronted with errors in the input.

Note: You should store all floating point numbers using the type `double`

Sample Test Data

To help you get started with testing we have created some sample input data for you to test the different stages of your program with.

In your ass2 directory type

```
cp ~dp1091/public_html/19T1/assignments/ass2/sampleFiles/* .
```

Note: You will also need to create your own test data to test your program thoroughly.

Viewer

The graphics output can be displayed as a graphical picture by using the viewer that has been provided. You don't need to use the viewer to implement the assignment (in fact it cannot be used for stages 0 and when using the -s command line option), but it's pretty boring without it.

- **Usage:** ~dp1091/bin/lineViewer < samplePic1.data

- You can optionally provide a command line argument to change the colour of the background, which by default is black by using:

```
~dp1091/bin/lineViewer white < samplePic1.data
```

Available background colours are white,red,green,blue

Command Line Arguments

The number of command line arguments and their values determine what stage of the assignment we are running.

- **No command line arguments:** If no command line arguments are supplied, your program should behave in the way described in **Stage 0**
- **First command line argument:**
When there are 1 or more command line arguments, the first argument will represent a value of n, which determines the number of iterations used in Stages 1 onwards (this is described in detail later). For stage 1 we will only ever test your code with a value of 0 for this command line argument. If there is only 1 command line argument, you should output the graphical data.
- **Second Command line argument:**
 - If there is a second command line argument "-s", instead of outputting graphical data you should output the results of your rewrite system, as outlined in stage 2.

Note: To get started on stage 0 you can simply ignore command line arguments for the time being.

You can assume we will only test with valid command line arguments, however your program should do something sensible in the case of incorrect command line arguments.

Implementing the string rewriting on C strings (arrays of characters), wouldn't work very well, since we have to insert characters in the middle of the string. You should therefore first implement your own string type based on **linked lists**, with all the operations necessary to perform rewriting on the strings. Do not use recursion to implement the rewriting.

Stage 0: [2 marks] Reading Input

In this stage you simply need to read in the user data from standard input and print it out on the screen with the appropriate format with all double values printed out to 4 decimal places.

For example assuming you used the sample input data supplied

```
./ass2 < inputData1.txt
Line Length: 10.0000
Initial Direction: 0.0000
Angle Increment: 0.7854
Initial Colour: RGB(0.0000,0.0000,1.0000)
Start String: F
Rules:
'F' -> "F+F+F+F+F+F+F"
```

```
./ass2 < inputData3.txt
Line Length: 100.0000
Initial Direction: 0.0000
Angle Increment: 1.5708
Initial Colour: RGB(1.0000,0.0000,0.0000)
Start String: F
Rules:
'C' -> "B"
'F' -> "F+F-CF-F+F"
```

Assumptions about the input format

There are no assumptions about the size of the start string you may need to read in.

You MUST use a linked list to store the start string. Failing to do so results in 0 marks for the assignment.

You may assume you will only need to read in strings that represent rules in the grammar of at most 100 characters (not including the newline character or the end of string null characters).

You should continue reading in rules until a maximum of 20 rules have been read in or until we reach the end of file (EOF or ctrl -d).

Note: You can assume during automarking that the input will always be in the same format and will be valid. However you should try to make sure your program does something sensible when it is confronted with errors in the input.

Note: You should store all floating point numbers using the type `double`

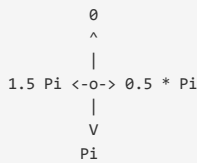
Stage 1 [3 marks] Generating graphical data

In this stage you need to implement a system that translates your linked list representation of a string of drawing commands, into a sequence of lines. You must convert your start string into some kind of linked list representation and translate that, otherwise your code will not be useful for further stages and you will not

receive any marks.

To do this your program needs to use the values you read in from standard input representing the

- Line Length in pixels
- Initial Direction given in radians ie



- Angle Increment: the angle by which + and - change the current direction, given in radians.
- Initial Colour: given in RGB format.

Your program must translate your string of drawing commands into a sequence of lines in a two-dimensional space. Each line is characterised by the following values:

- X- and Y-coordinate of the starting point
- X- and Y-coordinate of the end point point
- 3 double precision floating point values between 0 and 1 describing the color of the line as RGB (Red-Green-Blue) value. For example, (1.0 0.0 0.0) is red, (1.0 1.0 1.0) white, (0.0 0.0 0.0) black.

The output of your program should be the sequence of lines with each line in the format

```
xStart yStart xEnd yEnd R G B
```

The x and y values should be rounded down to integer values, and the RGB values are double precision floating point values.

RGB values should be incremented or decremented when appropriate, by a constant value of 0.1, however you need to make sure that the RGB values are never less than 0 and never more than 1.

Note: The starting point is always the origin (0,0).

To calculate the x and y co-ordinates you will need to do a little maths! To do this you will need to include the math.h library which contains functions such as `sin` and `cos`.

Floating point operations are a complex issue in computer science. To minimise floating point rounding errors and to ensure your solution's output is as close as possible to the required output, do the following:

- use double precision floating point values for all your internal calculations
- only convert the points to integer values as you output them
- use something like this for the conversion when you print:

```
printf ("%lld %lld ...", (long long) xStart, (long long) yStart,...);
```

For this stage, you can ignore the `[` and `]` commands. They will be implemented in in stage 3.

For this stage, you can also ignore the rewriting system. For this stage, you can assume you will always be given a command line argument of 0 (ie no rewrites).

In the following example we have a starting direction of 0 radians and a lineLength of 100 pixels. The start string is "F+F-F-F+F" and since there are no rewrites this is the final string that gets translated into a sequence of lines. Every time an `F` is encountered, the current position is printed out, along with the end point of the line if you travelled 100 pixels in the current direction and the current colour. Every time a `+` is encountered the current direction is incremented by the a given constant amount and whenever a `-` is encountered the current direction is decremented by a given constant amount. In this example there were no changes to the original colour.

```
./ass2 0 < inputData5.txt
0 0 0 100 1.000000 0.000000 0.000000
0 100 100 100 1.000000 0.000000 0.000000
100 100 100 200 1.000000 0.000000 0.000000
100 200 0 200 1.000000 0.000000 0.000000
0 200 0 300 1.000000 0.000000 0.000000
```

In the following example, the string is "FFFMF+BFFMGFMF+BFMG". In this example we have some characters that are meaningless to us when generating our drawing data, such as `M` and `T`. These should simply be ignored. We also have `B` and `G` commands which increase the levels of blue and green respectively in the current colour.

```
./ass2 0 < inputData6.txt
0 0 0 100 0.500000 0.200000 0.000000
0 100 0 200 0.500000 0.200000 0.000000
0 200 0 300 0.500000 0.200000 0.000000
0 300 0 400 0.500000 0.200000 0.000000
0 400 38 492 0.500000 0.200000 0.100000
38 492 76 584 0.500000 0.200000 0.100000
76 584 114 677 0.500000 0.300000 0.100000
114 677 153 769 0.500000 0.300000 0.100000
153 769 223 840 0.500000 0.300000 0.200000
```

To run your output from this stage through the lineviewer you could do something like:

```
./ass2 0 < inputData6.txt > pictureOutput6.data
~dp1091/bin/lineViewer < pictureOutput6.data
```

Or

```
./ass2 0 < inputData6.txt | ~dp1091/bin/lineViewer
```

Note: It is ok to have negative angles or angles greater than 2π . We will not test cases where the maximum angle will cause the double type to overflow or underflow, even after repeatedly performing '+' and '-' operations.

Stage 2 [4 marks] Grammar rewrite system

Your next task is to implement a rewrite system that applies the rules of the grammar to the given start string that you read in as part of stage 0. This is what can make our graphical output more interesting!
Rules in the grammar are of the form

```
'char1' -> "replaceStr1"
'char2' -> "replaceStr2"
etc
```

where *char1* is the character of the first rule, *replaceStr1* the string. So, when the the first rule is applied, all occurrences of *char1* in the start string get replaced by *replaceStr1*, and so on. If the grammar consists of multiple rules, start by applying the first rule to the start string, then apply the second rule to the resulting string, until all the rules have been applied once.

The program has to apply the rules to the resulting string until all rules have been applied *n* times, where *n* is given as a command line argument. It should write the resulting string to standard output.

For this stage you will be given an integer command line argument and then another optional command line argument of "-s"

The integer command line argument specifies how many iterations of rewrites you will do. The optional command line argument if it exists, indicates your output from the program should be the final output string rather than line output data.

In this example the value of *n* is 0 so we apply the rules 0 times (ie we do not apply the rules at all) and we print out the final string since we have the command line argument "-s".

```
./ass2 0 -s < inputData1.txt
StartString: F
After 0 iterations:
F
```

In this example the value of *n* is 1 so we apply the rules 1 time.

```
./ass2 1 -s < inputData1.txt
StartString: F
After 1 iterations:
F+F+F+F+F+F+F
```

In this example the value of *n* is 2 so we apply the rules 2 times.

```
./ass2 2 -s < inputData1.txt
StartString: F
After 2 iterations:
F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F
```

In this example our input file grammar has 2 rules. We must apply each rule one-by-one and in order.

```
./ass2 1 -s < inputData3.txt
StartString: F
After 1 iterations:
F+F-CF-F+F
```

In this example our input file grammar has 2 rules. We must apply each rule one-by-one and in order. We have a command line argument of 2 for *n*, so this process must be repeated.

```
./ass2 2 -s < inputData3.txt
StartString: F
After 2 iterations:
F+F-CF-F+F+F+F-F-CF-F+F-BF+F-CF-F+F-F+F-CF-F+F+F+F-F-CF-F+F
```

If there is no "-s" command line argument give, you must pass your final string generated by your rewrite system into your Graphical Data Generation System from Stage 1. For example

```
./ass2 2 < inputData3.txt
0 0 0 100 1.000000 0.000000 0.000000
0 100 100 100 1.000000 0.000000 0.000000
100 100 100 200 1.000000 0.000000 0.000000
100 200 0 200 1.000000 0.000000 0.000000
0 200 0 300 1.000000 0.000000 0.000000
0 300 100 300 1.000000 0.000000 0.000000
100 300 100 200 1.000000 0.000000 0.000000
100 200 200 200 1.000000 0.000000 0.000000
200 200 200 300 1.000000 0.000000 0.000000
200 300 300 300 1.000000 0.000000 0.000000
300 300 300 400 1.000000 0.000000 0.100000
300 400 400 400 1.000000 0.000000 0.100000
400 400 400 500 1.000000 0.000000 0.100000
400 500 300 500 1.000000 0.000000 0.100000
300 500 300 600 1.000000 0.000000 0.100000
300 600 200 600 1.000000 0.000000 0.100000
200 600 200 700 1.000000 0.000000 0.100000
200 700 100 700 1.000000 0.000000 0.100000
100 700 99 600 1.000000 0.000000 0.100000
99 600 0 600 1.000000 0.000000 0.100000
0 600 0 700 1.000000 0.000000 0.100000
0 700 99 700 1.000000 0.000000 0.100000
99 700 99 800 1.000000 0.000000 0.100000
99 800 0 800 1.000000 0.000000 0.100000
0 800 0 900 1.000000 0.000000 0.100000
```

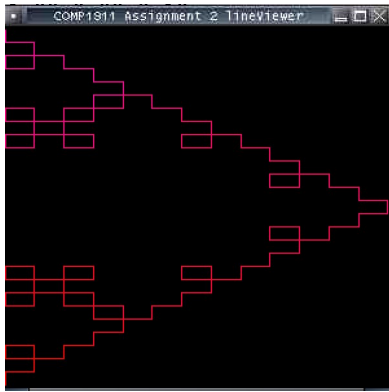
Running that data through the line viewer like this

```
./ass2 3 < inputData3.txt | ~dp1091/bin/lineViewer
```

or if you don't want to see all the line co-ordinate output and just want to see the picture:

```
./ass2 3 < inputData3.txt | ~dp1091/bin/lineViewer > /dev/null
```

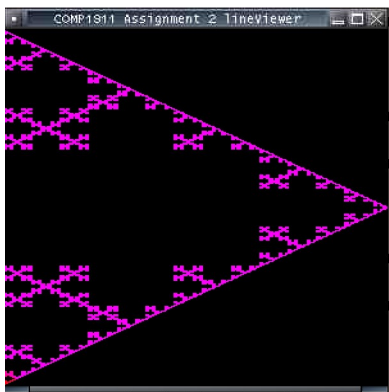
Should give you a picture that looks something like



Using more iterations such as:

```
./ass2 6 < inputData3.txt | ~dp1091/bin/lineViewer
```

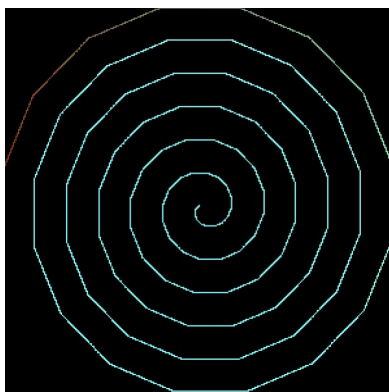
Should give you this:



And another example:

```
./ass2 100 < inputData4.txt | ~dp1091/bin/lineViewer > /dev/null
```

Should give you a picture that looks something like



Stage 3 [2 marks] Handling square brackets

For this stage you must extend your translation to deal with the square bracket commands. To implement this you need to be able to store the current position, direction and color when you encounter an opening square bracket, and restore this information when you encounter the matching closing bracket. The input and output format is the same as in the previous task.

Stage 3 differs from only in the fact that we will use data that contains '[' and ']'. It is run with the same command line arguments as stage2. Stage 1 and Stage 2 will NOT be tested with input with '[' or ']' characters. For example: stage 3:

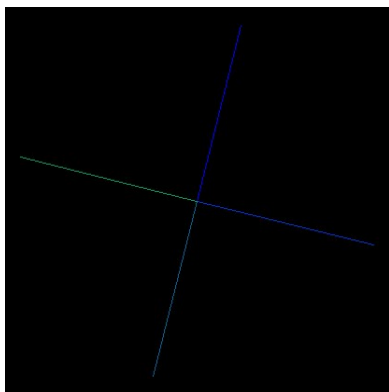
```
./ass2 1 -s < inputDataStage3_1.txt
StartString: [F]+[F]+[F]+[F]
After 1 iterations:
[F]G+b[F]G+b[F]G+b[F]G
```

```
./ass2 1 < inputDataStage3_1.txt
0 0 1 4 0.000000 0.000000 1.000000
0 0 4 -1 0.000000 0.100000 0.900000
0 0 -1 -4 0.000000 0.200000 0.800000
0 0 -4 1 0.000000 0.300000 0.700000
```

Running it as follows:

```
./ass2 2 < inputDataStage3_1.txt | ~dp1091/bin/lineViewer > /dev/null
```

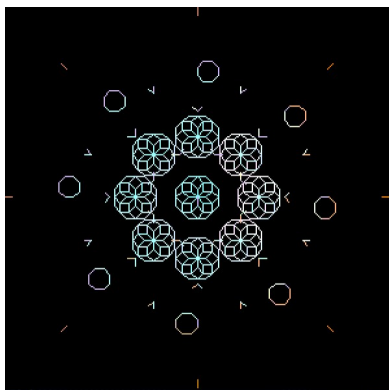
gives:



And another example:

```
./ass2 3 < inputDataStage3_4.txt | ~dp1091/bin/lineViewer > /dev/null
```

Should give you a picture that looks something like



[Click here to see the result of running](#)

```
./ass2 1 -s < inputDataStage3_4.txt
```

[Click here to see the result of running](#)

```
./ass2 1 < inputDataStage3_4.txt
```

Bonus Task : Art Gallery Competition [2 marks]

The language can be extended further, to produce more interesting pictures. For example, the line length as well as the angle could change. For this task, you can change the language any way you like, as long as the output of the interpreter still has the same line format. You can also change any of the limits that were imposed, so as a maximum of 20 rules in the grammar etc. Do your changes in a file called `ass2Bonus.c`. Do NOT make the changes in your `ass2.c` file. Submit the line file of the best picture you can come up with in a file name `"bonusPicture.txt"`, and make sure you have a description of your extensions in your blog along with the number of iterations you ran your program with to generate the picture. You should also submit the input you used in a file named `inputData.txt`.


Your tutor will award the bonus marks:

- up to 1 bonus marks for each submission (decided by your tutor, depending on the originality of your changes to the language & the beauty of the picture)
- 1.5 bonus marks for the best 2 entries of each tutorial
- 2 marks for the top 3 of the 'best in tutorial' entries (decided by vote on webcms)

Blog

You must keep notes on your blog every time you work on this assignment.

How to Blog

1. Go to the Course Webcms Home Page
2. Log in with your zid and zpass
3. Click on the speech bubble icon  at the bottom of the side navigation panel near your name.
4. Click on "Create Post"

Your blog should include

- The amount of time you spent
- Whether you were reading and understanding, designing, coding, debugging or testing.
- What you learnt or achieved
- Mistakes you made and how you could try to avoid or minimise them next time
- What kind of data structures you chose and why?. What other choices did you consider?

Hints:

- Be specific and include concrete examples of what you did. Only tutors can read your blog so you can include code snippets
- Don't panic if English is your second language or if you are not a 'great' writer. We understand and we do not expect the next great novel of our time.

Assessment

Assignment 2 is worth 15 Marks. Your submission will be assessed in the following way:

- Stage 0 (Automarking): 1 Marks
- Stage 1 (Automarking): 3 Marks
- Stage 2 (Automarking): 4 Marks
- Stage 3 (Automarking): 3 Marks
- Programming style: 3 Marks
- Assignment blog: 1 Mark
- Bonus: 2 Marks

Note: You can get 17/15 for this assignment if you do the bonus stage . You can get 15/15 without doing the bonus stage. Getting 17/15 can make up for lost marks in assignment 1 or labA marks. We'll test performance by running your program on a range of inputs (test cases). There will be separate tests for each of the stages of the assignment. Marks are awarded for each test case for which your program produces output identical to the output of the reference program. About half the marks will be for simple tests and about half for trickier ones.

Note: We do not just use the same tests as the submission autotests. We test on a whole batch of unknown test cases. Passing all the submission tests is a great start, but does not guarantee you will pass all of our test cases.

Remember your program is going to be automatically marked so be careful to follow the desired output to the letter ;)

Submit early so you can see the results of the submission autotests in case they reveal problems.

Submit the standard part of the assignment using this *give* command:

```
give dp1091 ass2 ass2.c
```

To just run the automarking tests without submission run

```
~dp1091/bin/autotest ass2
```

To submit for the bonus task

```
give dp1091 ass2Bonus ass2Bonus.c inputData.txt readMe.txt
```

Note: Your readMe.txt should tell us how to generate your picture. For example, how many iterations do we run your inputData.txt file with.

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 1%. For example if an assignment worth 76% was submitted 5 hours late, the late submission would have no effect. If the same assignment was submitted 30 hours late it would be awarded 70%, the maximum mark it can achieve at that time.

Late submissions will not be considered for Bonus marks