The University of Chicago
Department of Computer Science

CMSC 16200 – Honors Introduction to Computer Science 2
Winter Quarter 2007
Lab #7 (02/19/2007)
Due: 02/21 (Wed) at 5pm

Name:			
Student ID:		*Lab Instructor:*	Borja Sotomayor

Do not write in this area

Impl (20)	Style (10)	**TOTAL**

Maximum possible points: 30

This is your individual grading sheet. You must hand this sheet along with the group grading sheet and the individual grading sheets of everyone in your group.

The University of Chicago
Department of Computer Science

CMSC 16200 – Honors Introduction to Computer Science 2
Winter Quarter 2007
Lab #7 (02/19/2007)
Due: 02/21 (Wed) at 5pm

Name	***Student ID***

Lab Instructor: Borja Sotomayor

Do not write in this area

Design (10)	Doc (10)	Overall (10)	**TOTAL**

Maximum possible points: 30

This is your group grading sheet. You must hand this sheet along with the individual grading sheets of everyone in the group.

The University of Chicago
Department of Computer Science

CMSC 16200 – Honors Introduction to Computer Science 2
Winter Quarter 2007
Lab #7 (02/19/2007)
Due: 02/21 (Wed) at 5pm

Lindenmayer Systems

A Lindenmayer System, or L-System, is a formal grammar that can be used to model the growth of certain organisms. Starting from an L-System, it is possible to generate diagrams that are strikingly similar to real organisms (more details, and many beautiful examples, can be found in "The Algorithmic Beauty of Plants", by Przemyslaw Prusinkiewicz and Aristid Lindenmayer). In this lab, we will deal only with deterministic context-free L-Systems, which are similar to the grammars we have seen when studying parsers. The main difference is that the purpose of L-Systems is not to recognize strings, but to *generate* them.

An L-System is composed of the following:

- V: A set of variables (more formally, a set of symbols which can be replaced during the string generation)
- C: A set of constants (more formally, a set of symbols which remain fixed during generation)
- S: The start symbol (a symbol from V)
- P: A set of production rules. Like the production rules of parsing grammars, these rules have a predecessor (which, in the case of a context-free L-System, will be a single symbol from V) and a successor (a sequence of symbols from V and C).

For example, take the following L-System:

V: {A, B}
C: {+, –}
Start: A
Production rules:
A → B – A – B
B → A + B + A

Starting with A, we can generate strings indefinitely (*n* refers to the number of substitutions we have done):

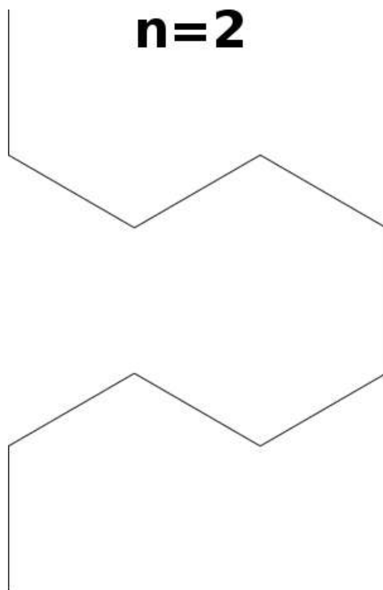
n=1 B – A – B
n=2 A + B + A – B – A – B – A + B + A
n=3 B – A – B + A + B + A + B – A – B – A + B + A – B – A – B – A + B + A – B – A – B + A + B + A + B – A – B
n=4 etc.

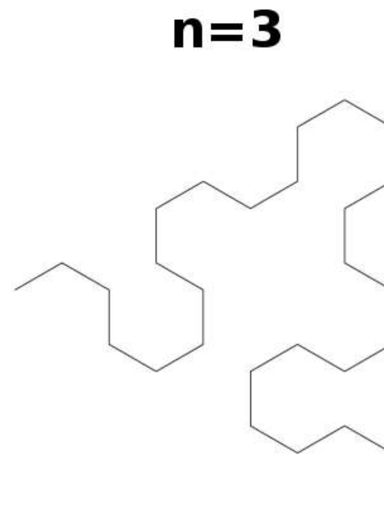
Of course, these strings by themselves might not make much sense. However, when interpreting each symbol as *drawing commands*, we can arrive at a graphical representation of the string.

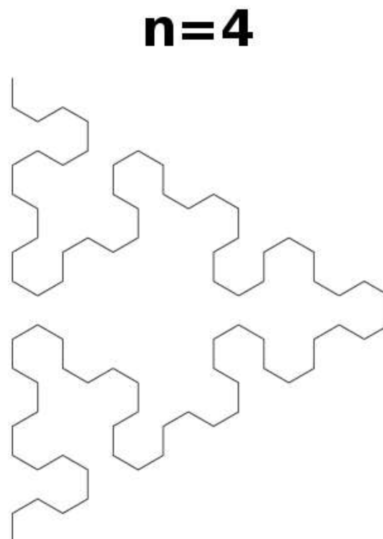
In particular, once we decide to stop the generation at some particular value *n*, we can interpret each symbol as follows:

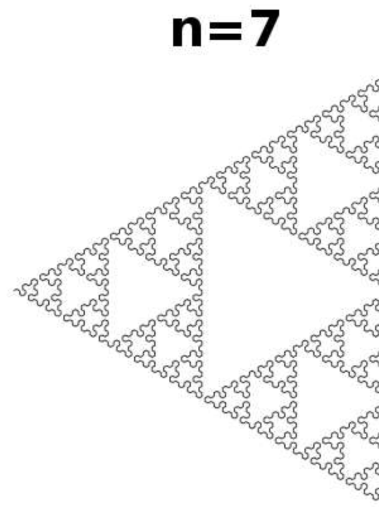
- A: Move forward one step (and draw a line)
- B: Move forward one step (and draw a line)
- +: Turn left by 60º
- -: Turn right by 60º

By doing this, we arrive at the following graphics:









In fact, the drawing actions required in the L-Systems we are dealing with map nicely to Turtle commands. The purpose of this lab will be to write a parser for arbitrary L-System grammars, and then generate a Turtle file with the drawing commands generated by the L-System.

The Grammar File

Our grammar files will be divided into three sections: assignments, production rules, and final production rules. This is an example of a grammar file for the L-System shown above:

```
limit = 7
angle = 60
start = A

%%

A -> B-A-B;
B -> A+B+A;

%%

A => “forward 1”;
B => “forward 1”;
```

The assignments section is used to specify *parameters* relevant to string generation:

- “limit”: The number *n* of replacements to perform.
- “angle”: The angle to turn when we encounter a + or a -.
- “start”: The start symbol.

Although only the above three values will be relevant to the generation process, our parser must be able to parse any parameter. Parameters names are composed only of lowercase letters, and contain at least one letter. Parameter values can be integers, real numbers, or the name of a variable.

The following section specifies the production rules in our system. The predecessor in a rule must be a variable name, which must be a single uppercase letter. The successor will be a sequence of variables and constants. The only accepted constants in our L-Systems are +, -, [, and] (the latter two are explained in the next section). The predecessor and successor are separated by “->”, and the end of a production rule is indicated with a semicolon.

The last section specifies the final production rules of our system. A final production rule specifies the Turtle command that is issued in place of a variable when string generation is complete. Note that we do not need to specify final production rules for constants, as those will have well-known default values. So, the predecessor can only be a variable

name. The predecessor and successor are separated by "=>". The successor of the rule is a string of characters delimited by double quotes. There is no restriction on what characters can be contained inside the double quotes (except the newline character). The end of a final production rule is delimited with a semicolon. If a final production rule requires more than one Turtle command to be issued, these will be separated by semicolons inside the double quotes (e.g., "forward 1; right 30; forward 1"). Note that, in the resulting Turtle file, each command must be placed in a separate line.

Each section in the grammar file is separated by "%%".

Generating and Visualizing the Turtle File

Given a grammar, generating the Turtle file is a simple process:

- Set *n*=1.
- Start at the production rule for symbol *start*.
- Traverse the symbols in the successor and do the following:

  If a symbol is a variable *v*, then:
  - if *n<limit,* increase *n* by one and traverse the symbols in replace it the successor of the production rule for *v*.
  - If *n=limit*, produce the Turtle command/s specified in the successor of the final production rule for *v.*

  If a symbol is a constant, then:
  - Constant +: Produce Turtle command "left *angle*"
  - Constant -: Produce Turtle command "right *angle*"
  - Constant [: Produce Turtle command "push"
  - Constant]: Produce Turtle command "pop"

To visualize the Turtle file, we are providing you with a set of Python scripts that take the Turtle file and generate a Postscript file with the resulting graphic. You can visualize this file with the *gv* command, or by converting it to a PDF file. We also provide a simple script, called *visulinden*, that will run all the scripts in the correct order to produce and visualize the Postscript file.

What you have to do

The result of this lab must be a single tool, called **lm2tg** that takes a grammar file, as described above, as its single parameter and uses it to generate a Turtle graphics file. The style of this lab is a *mini-project*. You must divide the work in the lab into components, which must be developed *individually* by each group member, and then pieced together at the end to produce a final result.

Just a spoonful of Software Engineering...

This might seem like a huge lab, but take into account that you will be dividing all the work amongst three people. The difficulty of this lab does not stem from the code you are asked to write, but in the *process* you will need to follow. This lab requires developing two major components: an L-System grammar parser, and an L-System processor. Each one is not too hard to write by itself, but making sure that both will work together is not a trivial endeavor if the work is divided amongst several programmers. For example, if you simply assigned each component to a different person, and they each represented the L-System internally using completely different data structures, piecing all the code together at the end can be a nightmare. To avoid this problem, this lab requires a good deal of planning and good design.

In particular, we suggest that you follow these steps:

- *Analysis*: In this step, you must come up with the requirements of the software project you are going to develop. This step has been largely done for you, but you must still make sure that you understand the requirements unambiguously. This is the step when you should ask the "client" for clarifications, if any of the stated requirements are not clear.
- *Project planning*: Divide the lab into different tasks and assign them to individual group members (a possible plan is provided below).
- *Design*: Before you can start writing any code, you will have to design the internal representation for an L-System. This includes not just an abstract representation, but also a concrete set of interfaces you will use to manipulate an L-System. It is very important that you all agree on this design: if anyone strays from the design during the implementation phase, all hell will break loose when you have to put all the different components together.
- *Unit implementation*: Implement each individual component in the project.
- *Integration*: Integrate components to create a fully-functional application.
- *Validation*: Check that the applications produce the desired result.

The following is a possible attack plan for this lab:

- Design tasks [All the group]
  - Design a grammar to parse the grammar file.
  - Design data structures to represent an L-System.
  - Design C interfaces to manipulate an L-System.
- Unit implementation
  - Implement an L-System parser. This component can be tested individually by checking if it can read the provided example files. Even if the functions to manipulate an L-System are not implemented yet, this component can still include calls to "dummy" functions (that implement the interfaces you all agreed on, but don't actually do anything).
  - Implement functions to manipulate the L-System (e.g., "create an L-System",

  "add a production rule", etc.)
  - Implement the generation algorithm. In this case, even if the parser is not implemented, this component can be tested by hardcoding a particular L-System into your code, and verifying that it generates the Turtle code correctly.
- Integration: Take the above three components and create a single program.
- Validation: Validate that your implementation correctly parses the provided example files, and produces graphics that are reasonably similar to the provided example graphics.

Restrictions

- The grammar file must be parsed with lex/yacc.
- All the components in your solution must be implemented in C.

Deliverables

- The **lm2tg** program. Make sure to include makefiles and instructions on how to compile. Also, please make sure you include no temporary files (object files, scratch files, etc.) in your handin.
- Documentation
  - At the end of the lab session (or end of Monday) hand in a quick description of the design of your application (data structures, C interfaces).
  - With the final handin, include a DOCUMENTATION.txt file explaining the final design of your application, and what modifications (if any) you had to make to your design as you developed your application.
  - You must explicitly state who was in charge of what tasks in the lab.

Grading

You will be graded both individually (30 points) and as a group (30 points). For the individual portion of the lab, you will be graded based on:

- *Correctness of your implementation* (20 points). Does your component do what it's supposed to do? Did you use the best possible solution to the problem at hand?
- *Style* (10 points): Is your code easy to read? Is it elegant?

For the group portion of the lab, every group member will receive the same grade based on the performance of the group as a whole:

- *Design* (10 points): Does the design meet the requirements specified in the lab?
- *Documentation* (10 points)
- *Overall* (10 points): Are the different components integrated correctly? Does the application work as expected?