

Project Due: Saturday, April 24, 8am
50 points

No LATE projects accepted after Thursday, April 29, 8am.

The purpose of this assignment is to write an interpreter for the LSYS programming language (see the project 1 and project 2 handouts for a description of the tokens and the grammar of the LSYS programming language). Your program will read in a data file containing a LSYS program, and if it is a syntactically correct LSYS program, then you will interpret the program and print JAWAA commands indicating the drawing of the lsystem. Your output can then be put on a web page to animate the program. See the JAWAA homepage for more details.

<http://www.cs.duke.edu/~wcp/jawaa.html>

DESCRIPTION OF YOUR PROGRAM

Given a sample LSYS program, your task is to 1) scan the program and identify all its *parts* (or *tokens*) 2) parse the program using an LR parser and identify if it is syntactically correct 3) construct a syntax tree and 4) “run” the LSYS program by traversing the syntax tree.

Part 1 - The Scanner

This was done in project 1.

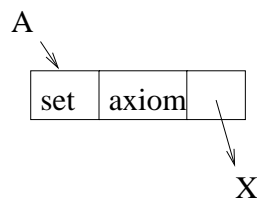
Part 2 - The Parser

This was done in project 2.

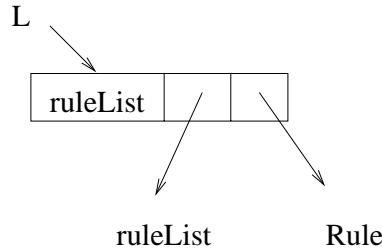
Part 3 - The Syntax Tree

For each LSYS program, you will construct a syntax tree that represents the semantics of the LSYS program. The tree can be built as the LSYS program is parsed.

Whenever structure is recognized in a LSYS program, the parts of the structure can be put together in the form of a syntax tree. Structure is recognized when a reduce operation is encountered. For example, when “set axiom X” is reduced to “Axim”, a syntax tree can represent the fact that X is the starting point in the grammar. We will create a node of type “set”. This node should contain a pointer to “X” in the symbol table.



For another example, when “ruleList Rule ;” is reduced to “ruleList”, there already exists a syntax tree for “ruleList” and a syntax tree for “Rule”, and they are joined together into one syntax tree for the new “ruleList” by creating a node of type “ruleList” (indicating a sequence of rules) containing pointers to the two syntax trees.



In order to keep track of the syntax trees, a stack called STstack will contain *pointers* to the current syntax trees and to variables in the symbol table. Whenever a reduce operation is encountered whose rewrite rule contains two nonterminals on the right hand side (representing two syntax trees that have previously been calculated), the top two pointers on the STstack are popped and joined together in a new syntax tree. Then the pointer to this new syntax tree is placed on the stack. Whenever a reduce operation is encountered whose rewrite rule contains one nonterminal on the right hand side, the top pointer on the STstack is popped and then pushed back onto the stack. Since this results in the STstack remaining the same, the stack does not need to be manipulated in this case. Whenever a reduce operation is encountered whose rewrite rule contains just terminals on the right hand side, a syntax tree node is created, pointers to the nonterminal's value in the symbol table are popped off of the STstack and placed into the syntax tree node, and then the pointer to the syntax tree node is pushed onto the STstack. When a LSYS program is recognized as valid, there will be one pointer on the STstack. This pointer points to the root of a syntax tree that represents the program. NOTE: the STstack is not the same stack the LR parser uses, but the two stacks do operate in parallel.

Types of nodes for syntax trees:

- *program* - This type of node represents the beginning of a LSYS program (the P rule) and has four parts. The first part tells the type of the node, *program*, the second part is a pointer to a set node (representing the set axiom), the third part is a pointer to a ruleList subtree, and the fourth part is a pointer to a DefList subtree. We don't need to represent the word "end" anywhere.
- *set* - This type of node has three parts. The first part tells that it is a set node. The second part tells what type of set node it is (axiom, color, length, angle, or width). The third part is a pointer to the symbol table, to a variable, a color or an integer (corresponds to what the second pointer was).
- *ruleList* - This type of node is a sequence node to connect a rule and a ruleList, and has three parts. The first part states that it is a ruleList node. The second part points to a ruleList, and the third part points to a rule node.
- *DefList* - This type of node is a sequence node to connect a definition and a DefList, and has three parts. The first part states that it is a DefList node. The second part points to a DefList, and the third part points to a set node (which represents a definition).
- *Rule* - This type of node has three parts. The first part tells the type of the node, *rule*, the second part points to a variable in the symbol table, and the third part is a pointer to an expression node.

- *leTter* - This type of node has three parts. The first part identifies the type of node, *leTter*. The second part is the type of leTter node (variable, symbol, or change) and the third part is a pointer to a color in the symbol table if the type is change, and otherwise a pointer to a variable or operator in the symbol table.
- *Expression* - This type of node has four parts. The first part tells the type of the node, *expression*, the second part tells the number of the expression rule (8, 9, 10, or 11), and the last two parts are pointers to the one or two fields needed to represent the expression.

Consider the following LSYS program.

```
// program 1
set axiom X

X ==> g ;

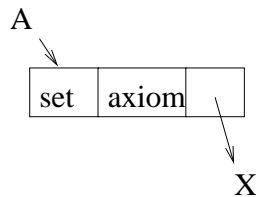
set length 10
end
```

This LSYS program can be derived by applying the following production rules (using the variable representations from project 2):

| RULES | DERIVATION |
|-------------------------|---|
| $P \rightarrow A L D e$ | A L D end |
| $D \rightarrow s F$ | A L set F end |
| $F \rightarrow l i$ | A L set length 10 end |
| $L \rightarrow R ;$ | A R ; set length 10 end |
| $R \rightarrow v a E$ | A X ==> E ; set length 10 end |
| $E \rightarrow T$ | A X ==> T ; set length 10 end |
| $T \rightarrow v$ | A X ==> g ; set length 10 end |
| $A \rightarrow s x v$ | set axiom X X ==> g ; set length 10 end |

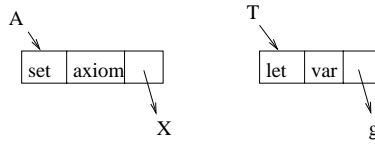
If we apply the rules in the reverse order (the order an LR parser would find them) we can construct the syntax tree for this LSYS program.

$A \rightarrow s x v$



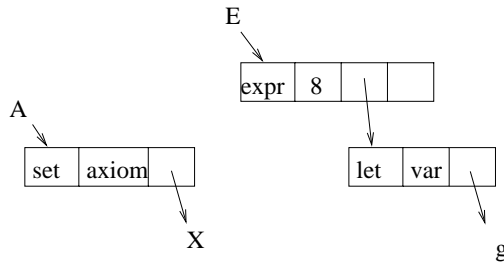
In this case, the pointer to the variable in the symbol table X is already on the STstack (it was pushed on during parsing when the lookahead is shifted onto the stack). It is popped off, the set node is created and a pointer to the set node is pushed on the STstack.

T → v



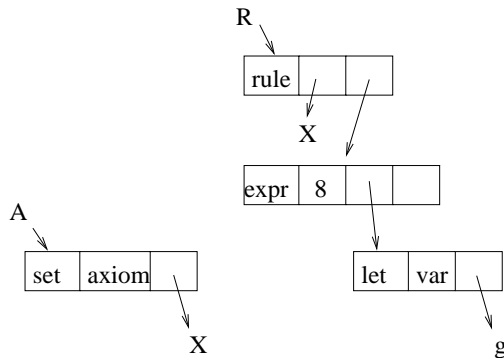
Here a letter node is created. The variable *g* is already on the STstack. It is popped off and put into the letter node. The second field of the node indicates that the letter is a variable. A pointer to the letter node is pushed on the STstack.

E → T



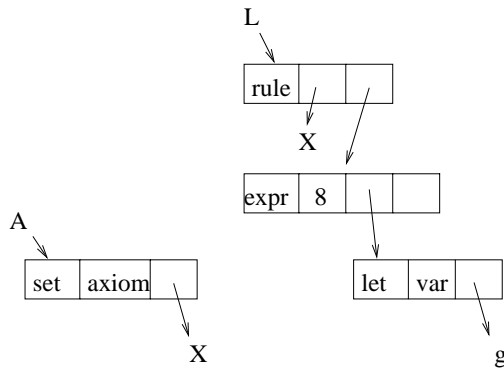
An expression node is created. The second field, 8, represents that rule 8 is the expression rule used. The third field points to a letter node (which is popped off the STstack) and the fourth field is not used in this case. A pointer to the expression node is then pushed on the STstack.

R → v a E



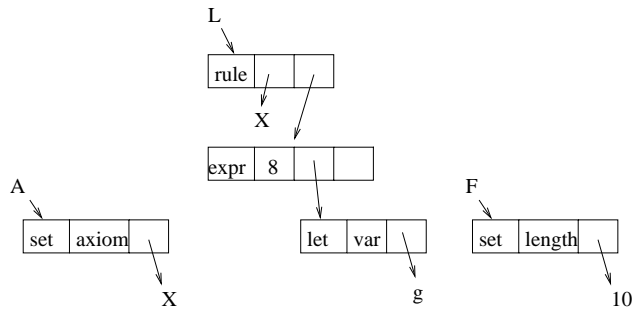
The variable (pointer to *X* in the symbol table) and the expression tree are popped off the STstack, a rule node is created and pushed back onto the STstack.

L → **R** ;



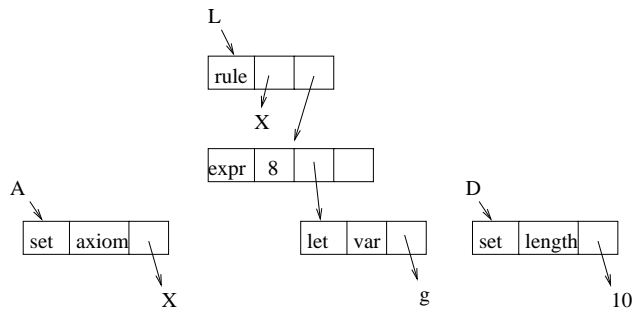
In this case, the rule is popped off the STstack and then pushed back onto the STstack (this pop and push can simply be ignored).

F → **l i**



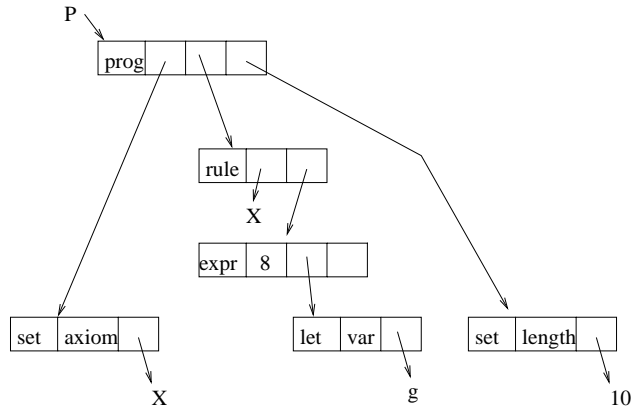
A pointer to 10 in the symbol table is popped off the STstack. A set node is created of type length, and pushed back on to the STstack.

D → **s F**



Here, one thing is popped off the stack and the same thing is pushed onto the stack, so this pop and push can be ignored.

P → **A L D e**



There are three pointers on the STstack that are popped. A program node is created, it uses the three pointers popped off, and then the program node is pushed onto the STstack. At this point there is one node on the STstack, a pointer to a tree that represents the complete LSYS program.

Part 4 - Execution of LSYS programs

If the parser identifies that the LSYS program is syntactically correct, then one can walk through the syntax tree and “run” the LSYS program.

We will have one lecture explaining the rules of interpreting an LSYS program.

In the example above, one would traverse the syntax tree and 1) use jawaa to create an initial window 2) a turtle would start in the lower middle of the screen and walk upwards 10 units (drawing a line) and then stop.

INPUT:

The input is a LSYS program. You may assume the tokens for LSYS programs are all valid. The format of the data file is the same as it was in projects 1 and 2.

OUTPUT:

Indicate whether the LSYS program is syntactically correct or not. If it is syntactically correct, then run the LSYS program and produce suitable output for JAWAA.

THE PROGRAM AND ITS SUBMISSION

Your program should be written in C++ and compile on the acpub or CS machines. (Use the g++ compiler).

Your program will be graded on style as well as content. Style will count for 20% of your grade.

Appropriate style for this course includes:

- *Modularity* - Your program should be divided into modules. Each module should have a single purpose that is described in a block comment at the beginning of the module.
- *Liberal use of comments* - In addition to the comment for each module, each nontrivial section of code (for example a loop) should have a comment describing its purpose. Comments should not merely echo the code.
- *Readability* - Your program should use the indentation and spacing appropriately to make it easily readable. Your comments should be clearly distinguishable from the code.

- *Appropriate variable names* - Give variables names that describe their function.
- *Understandable output* - Your program should indicate its input as well as its output in a clear and readable manner. Remember, the output from your program is the only indication that it works!

The remaining of your grade is based on meeting the specifications of the assignment. If you do not get your program correctly running, for partial credit you may generate output that identifies which part (functions) of your program are correctly working. This output must also be clearly understandable or no credit will be given!

Submitting

You must submit your .cc part of the program in one file called project3.cc. If you have several .cc files, put them all in one file and make sure they compile with the command: make project3.

You should create a file called README that contains your name, the amount of time the project took, and anyone you received help from. If you have multiple files, then you should use a Makefile. Submit your program by using the submit140 command. For example, suppose you have a makefile called *Makefile*, a C++ program called *project3.cc* and *project3.h*. To send these files, type

```
~rodder/bin/submit140 prog3 README Makefile project3.cc project3.h
```

where prog3 is the assignment name. This command works only on the acpub machines.

Programs should be submitted by the due date. You should read your mail regularly after submitting your project in case the grader cannot compile your program.

LATE PENALTIES

See the syllabus for the late penalty policy for programs.

No late programs will be accepted after Thursday, April 29, 8am!

EXTRA CREDIT (1 pt) Create a interesting LSYS program. Submit it with your program. You can also submit it early and I will put this on a web page.