PART 1: NOTE THIS PART IS DUE EARLIER!
Submit a SPLaFA program whose window must be no bigger than 600 by 600. It must have at least 30 statements and use at least one of each type of statement. See the submit section at the end of this handout. The point of this is that I will make available all these programs for you to use in testing your program. Submit a different program than you submitted for project 2.

PART 2:
The purpose of this assignment is to write an interpreter for the SPLaFA programming language (see the project 1 and project 2 handouts for a description of the tokens and the grammar of the SPLaFA programming language). Your program will read in a data file containing a SPLaFA program, and if it is a syntactically correct SPLaFA program, then you will interpret the program and output JAWAA commands into a .anim file indicating what should be drawn and where it should be drawn or moved. More info on JAWAA is at the end of this handout.

(Alternatively, you can do the graphics in Java if you want, but JAWAA will be much easier.)

DESCRIPTION OF YOUR PROGRAM
Given a sample SPLaFA 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 SPLaFA program by traversing the syntax tree and producing an animation.

Part 1 - The Scanner
This was done in project 1.

Part 2 - The Parser
This was done in project 2. Remove all output statements from this part.

Part 3 - The Syntax Tree
For each SPLaFA program, you will construct a syntax tree that represents the semantics of the SPLaFA program. The tree can be built as the SPLaFA program is parsed.

Whenever structure is recognized in a SPLaFA 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 "move bob 10 skip" is reduced to "Statement", a syntax tree can represent the fact that the object bob should move 10 spaces to the right and $x$ spaces down, where $x$ is the value of the variable skip. We will create a node of type "move". This node should contain references to "bob" in the symbol table, to 10 in the symbol table and to "skip" in the symbol table.

For another example, when "List Statement ;" is reduced to "List", there already exists a syntax tree for "List" and a syntax tree for "Statement", and they are joined together into one syntax tree for the new "List" by creating a node of type "seq" (indicating a sequence of statements) containing references to the two syntax trees.

In order to keep track of the syntax trees, a stack called STstack will contain references to the current syntax trees and to variables in the symbol table. Whenever a reduce operation is encountered whose rewrite rule contains two items on on the right hand side with values already on the STstack (an item may be a nonterminal representing a syntax tree on the STstack, or a terminal with a reference on the STstack that points to the terminal in the symbol table), the top two references on the STstack are popped and joined together in a new syntax tree node. Then the reference to this new syntax tree is placed on the stack. Whenever a reduce operation is encountered whose rewrite rule contains one terminal or nonterminal on the right hand side, the top reference 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, references to the terminal’s value in the symbol table are popped off of the STstack and placed into the syntax tree node, and then the reference to the syntax tree node is pushed onto the STstack. Note there are some terminals that do not have values in the
symbol table (such as [ ]) and they do not have a value pushed onto the STstack. When a SPLaFA program is recognized as valid, there will be one reference on the STstack. This reference 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. When a lookahead that is also in the symbol table (such as a variable) is pushed onto the parsing stack, then a reference to it in the symbol table is pushed onto the STstack. If a lookahead is not in the symbol table (such as the keyword move), then when the lookahead is pushed onto the parsing stack, it DOES NOT have an item pushed down on the STstack.

Types of nodes for syntax trees:

- **program** - \(<size> [<list>]\) - This type of node represents the beginning of a SPLaFA program and has three parts. The first part tells the type of the node, program, the second part is a reference to a size node, and the third part is a reference to a list of statements, either a seq node if there are multiple statements, or a single statement node.

- **size** \(i\) \(j\) - This type of node has three parts. The first part tells the type of the node, size, and the second and third parts are references to integers in the symbol table.

- **sequence** - This type of node has three parts. The first part identifies the type of node, seq. The second and third parts are references to syntax trees, where those statements in the second reference’s syntax tree should be executed before those statements in the third reference’s syntax tree.

- **line** \(v\) \(a\) \(b\) \(c\) \(d\) \(color\) - This type of node has seven parts. The first part tells the type of the node, line, the second part is a reference to \(v\) in the symbol table, the third through sixth parts are references to integers or variables in the symbol table, and the last part is a reference to a color in the symbol table.

- **rect** \(v\) \(a\) \(b\) \(c\) \(d\) \(color\) - This type of node has seven parts. The first part tells the type of the node, rect, the second part is a reference to \(v\) in the symbol table, the third through sixth parts are references to integers or variables in the symbol table, and the last part is a reference to a color in the symbol table.

- **move** \(v\) \(a\) \(b\) - This type of node has four parts. The first part tells the type of the node, move, the second part points to the variable \(v\) in the symbol table, and the third and fourth parts are references to either integers or variables in the symbol table.

- **v = a** - This type of node has three parts. The first part identifies the type of node, asgn. The second part is a reference to the variable \(v\) in the symbol table, and the third part is a reference to \(a\) in the symbol table. \((a\) is a variable or integer).

- **for** \(v = a\) to \(b\) by \(c\) do \(<stmts>\) - This type of node has six parts. The first part identifies the node as a for node. The second part is a reference to variable in the symbol table. The third through fifth parts are references to either variables or integers in the symbol table. The sixth part is reference to a syntax tree that represents the body of the for
statement. The meaning of the for statement is to set \( v \) equal to \( a \). If \( v \leq b \) then execute the statements in the body. Repeatedly increment \( v \) by \( c \) and execute the statements in the body until \( v > b \).

Consider the following SPLaFA program.

```plaintext
data 300 200 [  
  skip = 5 ;  
  rect fred 10 20 40 70 red ;  
  move fred skip 0 ;  
]
```

This SPLaFA program can be derived by applying the following production rules (using the first letter of each variable):

<table>
<thead>
<tr>
<th>RULES</th>
<th>DERIVATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \rightarrow Z [ L ] )</td>
<td>( Z [ L ] )</td>
</tr>
<tr>
<td>( L \rightarrow L S ; )</td>
<td>( Z [ L S ; ] )</td>
</tr>
<tr>
<td>( S \rightarrow move var T T )</td>
<td>( Z [ L move fred T T ; ] )</td>
</tr>
<tr>
<td>( T \rightarrow int )</td>
<td>( Z [ L move fred T 0 ; ] )</td>
</tr>
<tr>
<td>( T \rightarrow var )</td>
<td>( Z [ L move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( L \rightarrow L S ; )</td>
<td>( Z [ L S ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( S \rightarrow rect var T T T c ; )</td>
<td>( Z [ L rect fred T T T T red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( T \rightarrow int )</td>
<td>( Z [ L rect fred T T 70 red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( T \rightarrow int )</td>
<td>( Z [ L rect fred T 40 70 red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( T \rightarrow int )</td>
<td>( Z [ L rect fred T 20 40 70 red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( T \rightarrow int )</td>
<td>( Z [ L rect fred 10 20 40 70 red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( L \rightarrow S ; )</td>
<td>( Z [ S ; rect fred 10 20 40 70 red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( S \rightarrow var = T )</td>
<td>( Z [ skip = T ; rect fred 10 20 40 70 red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( T \rightarrow int )</td>
<td>( Z [ skip = 5 ; rect fred 10 20 40 70 red ; move fred skip 0 ; ] )</td>
</tr>
<tr>
<td>( Z \rightarrow size int int )</td>
<td>( size 300 200 [ skip = 5 ; rect fred 10 20 40 70 red ; move fred skip 0 ; ] )</td>
</tr>
</tbody>
</table>

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 SPLaFA program.
\[ Z \rightarrow \text{size int int} \]

In this case, the two references to 300 and 200 in the symbol table are on the STstack. A size node is created, the references to 300 and 200 are popped and put into the size node, and a reference to this size node is pushed on the STstack.

\[ T \rightarrow \text{int} \]

NOTE: What does the STstack look like at this point?

In this case, the reference to the symbol table for 5 is already on the STstack. So is the reference to the variable skip. They were both pushed onto the STstack at the same time their tokens were shifted onto the parsing stack. Here when the reduction \( T \rightarrow \text{int} \) is encountered, you can pop the reference to 5 off the stack and then push it back on, or just do nothing.

\[
\begin{align*}
\rightarrow & \ 5 \\
\rightarrow & \ \text{skip} \\
\rightarrow & \ \text{size 300 200}
\end{align*}
\]
\( S \rightarrow \text{var} = T \)

In this case, a node of type \( \text{asgn} \) is created, the two references on the STstack are popped off the stack and put in this node, and then a reference to this node is pushed onto the STstack.

NOTE: What does the STstack look like at this point?

\[
\rightarrow \text{asgn} \text{ (which points to skip and 5)} \\
\rightarrow \text{size 300 200}
\]

\( L \rightarrow S \);  
Here note that the STstack does not change. You could pop and push the same thing back, or do nothing.
Four rules all the same: \( T \rightarrow \text{int} \)

\[
\begin{array}{c}
\text{Z} \\
\text{size} \\
300 \\
200 \\
\end{array}
\]

\[
\begin{array}{c}
\text{L} \\
\text{asgn} \\
\text{skip} \\
5 \\
\end{array}
\]

\[
\begin{array}{c}
\text{T} \\
\text{T} \\
\text{T} \\
\text{T} \\
\end{array}
\]

NOTE: What does the STstack look like at this point? (note fred is also on the stack)

\[ \rightarrow 70 \]
\[ \rightarrow 40 \]
\[ \rightarrow 20 \]
\[ \rightarrow 10 \]
\[ \rightarrow \text{fred} \]
\[ \rightarrow \text{asgn} \text{ (which points to skip and 5)} \]
\[ \rightarrow \text{size 300 200} \]

\[
\begin{array}{c}
\text{S} \\
\text{rect var T T T T c} \\
\text{size} \\
300 \\
200 \\
\end{array}
\]

\[
\begin{array}{c}
\text{L} \\
\text{asgn} \\
\text{skip} \\
5 \\
\end{array}
\]

\[
\begin{array}{c}
\text{red} \\
\text{S} \\
\text{rect} \\
\text{fred} \\
10 \\
20 \\
40 \\
70 \\
\end{array}
\]

NOTE: What does the STstack look like at this point?

\[ \rightarrow \text{rect node} \]
\[ \rightarrow \text{asgn node} \]
\[ \rightarrow \text{size 300 200} \]
\[ L \rightarrow L \ S \ ; \]

\[ T \rightarrow \text{var} \quad \text{and} \quad T \rightarrow \text{int} \]

\[ L \rightarrow \text{seq} \]

\[ \text{asgn} \quad \text{rect} \]

\[ \text{skip} \quad 5 \]

\[ \text{fred} \quad 10 \quad \text{red} \]

\[ 20 \quad 40 \quad 70 \]

\[ \text{size} \]

\[ 300 \quad 200 \]

\[ \text{size} \]

\[ 300 \quad 200 \]
\[
S \rightarrow \text{move var } T \ T \\
Z
\]

\[
\begin{align*}
\text{size} & \quad 300 & 200 \\
\text{move} & \quad \text{asgn} & \quad \text{rect} \\
& \quad \text{skip} & \quad \text{fred} & \quad \text{red} \\
& \quad 5 & \quad 10 & \quad 20 & \quad 40 & \quad 70
\end{align*}
\]

\[
L \rightarrow L S ;
\]

\[
\begin{align*}
\text{size} & \quad 300 & 200 \\
\text{seq} & \quad \text{asgn} & \quad \text{rect} \\
& \quad \text{skip} & \quad \text{fred} & \quad \text{red} \\
& \quad 5 & \quad 10 & \quad 20 & \quad 40 & \quad 70
\end{align*}
\]
Part 4 - Execution of SPLaFA programs

If the parser identifies that the SPLaFA program is syntactically correct, then one can walk through the syntax tree and “run” the SPLaFA program. When running a program, the current value of variables are stored in the symbol table. In project 1, each variable in the symbol table had an integer value associated with it that was initially set to 0. You may need to keep more information associated with your variables to run a program. For example a variable that is a rectangle, what information do you need to store that you will refer to later?

In the example above, one would traverse the syntax tree and 1) use the animator to create the initial window of size 300 by 200 (show this window) 2) assign skip in the symbol table the value 5, 3) create a rectangle named fred with left corner in position (10,20), and 4) then move the rectangle fred to position 5 units to the right.

The corresponding JAWAA code that would be generated for this animation would be:

```
rectangle window 0 0 300 200 black white
rectangle fred 10 20 40 70 red red
moveRelative fred 5 0
```
The first rectangle generated defines the window region for the animation. For this window, output it as a white rectangle with black outline whose top left corner is at position (0,0).
The second rectangle generated is the rectangle fred. Note in SPLaFA there is only one color for the rectangle. In JAWAA you must list the color twice as it will represent the outline and filled in color of the rectangle. Note the assignment statement does not produce any JAWAA output. Instead when the move command is interpreted, the JAWAA moveRelative command is generated with the value 5 (look up the value of skip in the symbol table and print 5 instead of skip).

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

OUTPUT:
If the SPLaFA program is syntactically correct, then run the SPLaFA program and produce suitable output for JAWAA to execute the program (a .anim file).

THE PROGRAM
Your program should be written in Java 5 or C++ and use Eclipse.

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 of your program works correctly. This output must also be clearly understandable or no credit will be given!

**SUBMIT: PART 1 NOTE: DUE EARLIER!**
Name your SPLaFA program with your login id and .splafa. For example if your login is mylogin, then your SPLaFA program should be named mylogin.splafa and should be submitted through ambient under project3Part1.

**SUBMIT: Part 2**

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 did the extra credit, you should mention this in the README file.

Submit your program by using ambient with the name project3Part2.

Programs should be submitted by midnight (23:59:59) on the due date.

**EXTRA CREDIT: Error Handling (5 pts)**

The extra credit must be turned in with your program (Part 2) on time. You can not submit the extra credit later.

You should handle files that contain the following semantic errors. Be sure to print informative error messages in your animation. You can display text (see the JAWAA text command) delay and then remove the message, or make the message float off the screen by moving it far off to the right.

- (4 pts) Print an error message if an object (line or rectangle) has coordinates outside of the window. You should be able to see this in the JAWAA window.

- (1 pt) Print an error message if a variable is used for which an assignment statement has not yet been executed. In this case, use 0 as the value for the variable and continue executing. Do flash a message in JAWAA to indicate that an error occurred.

**LATE PENALTIES**

Project 3 can be turned in by midnight on the following days with the following penalties:

- Sunday, April 22: 5pts off
- Tuesday April 24: 10pts off

See me immediately if the project is not complete by April 24!

**JAWAA - Getting Started**

To run a JAWAA animation, you create two files with the same name but different extensions. For example, sample.anim and sample.html. The .anim file has the JAWAA commands generated from your program. The .html file has the applet to run the animation.

To get started, you will create a cps140 directory that is visible to the web (in your public.html folder in your duke web space), and copy the sample files sample.anim and sample.html from the assignment page. This sample.html page has a link to the applet from the Jawaa web space. Load the .html file and you should be able to run the animation.
http://www.duke.edu/~yourlogin/cps140/sample.html

If it doesn’t work, then load the jawaa.zip file and unpack it. It should have a jawaa folder, sample.anim (same file as before) and a sample.html which has the applet linked to this jawaa folder. Now try loading this .html file and run it.

The animation creates a red rectangle and moves it 5 units to the right. You can control the speed of the animation with the speed control bar.

**JAWAA with Project 3**

In this project, your program should write JAWAA output to a separate file that only has JAWAA commands in it. Name this program with a .anim extension. Suppose you name it splafa1.anim.

To run this program, copy the splafa1.anim file to your public_html/cps140 directory. You also need an html page with the same name that has an applet to run JAWAA. For this you can copy the web page sample.html to splafa1.html, you don’t need to modify this file but if you want you can add some html to it (title, your name, etc.) .

**NOTES:**

- Your .anim and .html files must have the same name, different extensions.
- Your .anim and .html files must be in the directory where you have installed JAWAA, in your public_html/cps140 directory.
- Every time you make a change in your project 3, you must run your project 3, copy the .anim file to your public_html/cps140 directory, and reload the web page.

See the JAWAA web page for more info on the JAWAA commands and their format.

www.cs.duke.edu/csed/jawaa2