PART 1: SAPL Program Due: Thursday, April 10, 11:59pm
PART 2: SAPL Program for Contest Due: Thursday, April 17, 11:59pm
PART 3: Project Due: Thursday, April 17, 11:59pm
(55 points)

No LATE projects accepted after Tuesday, April 22, 11:59pm.

PART 1: NOTE THIS PART IS DUE EARLIER!
Submit a SAPL program. 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.

PART 2: NOTE THIS IS DUE THE SAME TIME AS PART 3!
Submit a second SAPL program, its .anim file and the location of its animation for a contest. It must have at least 40 statements and use at least one of each type of statement. See the submit section at the end of this handout. You will vote for the best program/animation!

PART 3: NOTE THIS IS DUE THE SAME TIME AS PART 2!
The purpose of this assignment is to write an interpretor for the SAPL programming language (see the project 1 and project 2 handouts for a description of the tokens and the grammar of the SAPL programming language). Your program will read in a data file containing a SAPL program, and if it is a syntactically correct SAPL 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.

DESCRIPTION OF YOUR PROGRAM
Given a sample SAPL 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 SAPL 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 SAPL program, you will construct a syntax tree that represents the semantics of the SAPL program. The tree can be built as the SAPL program is parsed.

Whenever structure is recognized in a SAPL 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 pointers 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 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 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 pointer on the STstack that points to the terminal in the symbol table), the top two pointers on the STstack are popped and joined together in a new syntax tree node. Then the pointer 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 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
erule contains just terminals on the right hand side, a syntax tree node is created, pointers to
the terminal’s value in the symbol table are popped o of the STstack and placed into the
syntax tree node, and then the pointer 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 SAPL 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. When a lookahead that is also in the symbol table (such as a variable) is pushed
onto the parsing stack, then a pointer 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 SAPL
  program and has three parts. The first part tells the type of the node, *program*, the
  second part is a pointer to a size node, and the third part is a pointer 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 pointers 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 pointers to syntax trees, where those statements
  in the second pointer’s syntax tree should be executed before those statements in the
  third pointer’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 pointer to *v* in the symbol table, the third through
  sixth parts are pointers to integers or variables in the symbol table, and the last part
  is a pointer 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 pointer to *v* in the symbol table, the third through
  sixth parts are pointers to integers or variables in the symbol table, and the last part
  is a pointer 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 pointers 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 pointer to the variable *v* in the symbol table, and the third
  part is a pointer to *a* in the symbol table. (*a* is a variable or integer).
• *for v = a to b by c do* `<stmtrs>` - This type of node has six parts. The first part identifies the node as a *for* node. The second part is a pointer to variable in the symbol table. The third through fifth parts are pointers to either variables or integers in the symbol table. The sixth part is pointer 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 ≤ 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 SAPL program.

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

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

**RULES**

<table>
<thead>
<tr>
<th><strong>DERIVATION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P → Z [ L ]</strong></td>
</tr>
<tr>
<td><strong>L → L S ;</strong></td>
</tr>
<tr>
<td><strong>S → move var T T</strong></td>
</tr>
<tr>
<td><strong>T → int</strong></td>
</tr>
<tr>
<td><strong>T → var</strong></td>
</tr>
<tr>
<td><strong>L → L S ;</strong></td>
</tr>
<tr>
<td><strong>S → rect var T T T c ;</strong></td>
</tr>
<tr>
<td><strong>T → int</strong></td>
</tr>
<tr>
<td><strong>T → int</strong></td>
</tr>
<tr>
<td><strong>T → int</strong></td>
</tr>
<tr>
<td><strong>T → int</strong></td>
</tr>
<tr>
<td><strong>L → S ;</strong></td>
</tr>
<tr>
<td><strong>S → var = T</strong></td>
</tr>
<tr>
<td><strong>T → int</strong></td>
</tr>
<tr>
<td><strong>Z → size int int</strong></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 SAPL program.
In this case, the two pointers to 300 and 200 in the symbol table are on the STstack. A size node is created, the pointers to 300 and 200 are popped and put into the size node, and a pointer to this size node is pushed on the STstack.

In this case, the pointer to the symbol table for 5 is already on the STstack. So is the pointer 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 → int is encountered, you can pop the pointer to 5 off the stack and then push it back on, or just do nothing.

→ 5
→ skip
→ size 300 200

NOTE: What does the STstack look like at this point?
S → var = T

In this case, a node of type `asgn` is created, the two pointers on the STstack are popped off the stack and put in this node, and then a pointer to this node is pushed onto the STstack.

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

→ `asgn` (which points to `skip` and 5)
→ `size` 300 200

L → 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: \( T \rightarrow \text{int} \)

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

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

\[
\begin{align*}
S \rightarrow & \text{rect var T T T T c} \\
Z & \\
L & \\
S & \\
\end{align*}
\]

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

\[
\begin{align*}
&\rightarrow \text{rect node} \\
&\rightarrow \text{asgn node} \\
&\rightarrow \text{size 300 200}
\end{align*}
\]
L → L S ;
Z

L

size

300 200
<asgn
<rect

T → var and T → int
Z

T

T

seq

asgn

rect

skip

fred

red

20

70

20

40

10

8
\[ S \rightarrow \text{move var } T \ T \]

\[ L \rightarrow L \ S ; \]

\[ S \rightarrow \text{move} \]

\[ Z \rightarrow \text{size} \]

\[ \text{rect} \]

\[ \text{skip} \]

\[ \text{asgn} \]

\[ \text{fred} \]

\[ \text{red} \]

\[ \text{10} \]

\[ \text{20} \]

\[ \text{40} \]

\[ \text{70} \]
Part 4 - Execution of SAPL programs

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

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

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

THE PROGRAM
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 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 SAPL program with your login id and .sapl. For example if your login is mylogin, then your SAPL program should be named mylogin.sapl and should be submitted by typing:

    ~rodger/bin/submit140 sapl mylogin.sapl

SUBMIT: PART 2
THIS SAPL program should have at least 40 statements. You should also submit a text file called website that includes your name and the http web address of your animation for this SAPL program. Name your SAPL program with your login id, the number 2 and .sapl extension. For example if your login is mylogin, then your SAPL program should be named mylogin2.sapl. The .anim file should have the same name. All three files should be submitted by typing:

    ~rodger/bin/submit140 saplcontest website mylogin2.sapl mylogin2.anim

SUBMIT: Part 3
You should use a Makefile. 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 the submit140 command. Submit all your .h and .cpp files. For example, suppose you have a README file, a makefile called Makefile, a C++ program called project3.cpp and project3.h. To send these files, type

    ~rodger/bin/submit140 pro3 README Makefile project3.cpp project3.h

where project3 is the assignment name. This command should work on the CS machines and on the acpub machines.

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 3) 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 20: 5pts off
- Tuesday April 22: 10pts off

See me immediately if the project is not complete by April 22!

JAWAA - Getting Started

To run a JAWAA animation, you create two files with the same name but different extensions. For example, `prog1.anim` and `prog1.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 directory that is visible to the web, and copy JAWAA and sample files. To do this type the following (don’t forget the dot at the end of the `cp` line):

```
cd
cd public_html
mkdir cps140
cd cps140
cp ~roger/cps140/jawaa/tarjawaa .
ls
```

If you did this correctly, you should see the one file `tarjawaa`. Then unpack all the files by typing:

```
tar xfv tarjawaa
```

This should create a `jawaa` directory and a sample html and anim file for you called `proj3a.html` and `proj3a.anim`. This is the generated JAWAA code for the SAPL program on page 4 of this handout.

You can now run this JAWAA animation by loading the following page (replace `yourlogin` with your login id) in your browser. You must use Netscape 6.0 or higher, or Internet Explorer.

```
http://www.duke.edu/~yourlogin/cps140/proj3a.html
```
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 `sapl1.anim`.

To run this program, copy the `sapl1.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 `proj3a.html` to `sapl1.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`