PART 1: ROBOGO Program Due: Tuesday, April 9, 11pm
PART 2: Project Due: Thursday, April 18, 11pm
33 points

No LATE projects accepted after Monday, April 22, 11pm.

PART 1: NOTE THIS PART IS DUE EARLIER!
Submit a ROBOGO program. It must have at least 30 statements and use at least one of each type of statement.

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

www.cs.duke.edu/~rodger/tools/jawaa/

NOTE: There was a typo on project 2. The two integers in the begin statement represent the size of the room the robots and obstacles will be in. The statement that starts with begin 60 80 means the width is 60 units and the height is 80 units.

DESCRIPTION OF YOUR PROGRAM
Given a sample ROBOGO 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 ROBOGO 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 ROBOGO program, you will construct a syntax tree that represents the semantics of the ROBOGO program. The tree can be built as the ROBOGO program is parsed.
Whenever structure is recognized in a ROBOGO 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 east skip” is reduced to “Statement”, a syntax tree can represent the fact that the robot bob should move x spaces in the direction east, 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 a node containing the direction “east” (created earlier) and to “skip” in the symbol table.

```
move
   
bob  east  skip
```

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.

```
seq
   
syn. tree
for
List
```
```
syn. tree
for
Statement
```

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 ROBOGO 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:

- **begin** - *begin i j <list> halt* - This type of node represents the beginning of a ROBOGO program and has four parts. The first part tells the type of the node, *begin*, the second and third parts are pointers to the integers *i* and *j* in the symbol table, and the fourth part is a pointer to a list of statements, either a *seq* node if there are multiple statements, or a single statement node.

- **robot v a b** - This type of node has four parts. The first part tells the type of the node, *robot*, the second part is a pointer to *v* in the symbol table, and the third and fourth parts are pointers to *a* and *b* in the symbol table. (*a* and *b* are integers or variables).

- **obstacle a b** - This type of node has three parts. The first part tells the type of the node, *obst*, and the second and third parts are pointers to *a* and *b* in the symbol table. (*a* and *b* are integers or variables).

- **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.

- **add a to v** - This type of node has three parts. The first part tells the type of the node, *add*, and the second and third parts are pointers to *a* and *v* in the symbol table. (*v* is a variable, and *a* is an integer or variable).

- **move v d a** - 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, the third part points to a node containing the direction, and the fourth part is a pointer to *a* in the symbol table. (*a* is an integer or variable).

- **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).

- **do <stmts> until a > b** - This type of node has four parts. The first part identifies the node as a *do* node. The second part is a pointer to a syntax tree that represents the body of the do statement. The third and fourth parts are pointers to *a* and *b* in the symbol table. (*a* and *b* are integers or variables). The meaning of the do statement is to execute the statements in the body first. If *a* > *b* then halt, otherwise repeat.

Consider the following ROBOGO program.

```plaintext
*-- program 1 --*
begin 40 60
    obstacle 7 11 ;
    robot bob 5 10 ;
    move bob east 6 ;
halt
```
This ROBOGO 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 → begin int int L halt</td>
<td>begin 40 60 L halt</td>
</tr>
<tr>
<td>L → L S ;</td>
<td>begin 40 60 L S ; halt</td>
</tr>
<tr>
<td>S → move var D T</td>
<td>begin 40 60 L move bob D T ; halt</td>
</tr>
<tr>
<td>T → int</td>
<td>begin 40 60 L move bob D 6 ; halt</td>
</tr>
<tr>
<td>D → east</td>
<td>begin 40 60 L move bob east 6 ; halt</td>
</tr>
<tr>
<td>L → L S ;</td>
<td>begin 40 60 L S ; move bob east 6 ; halt</td>
</tr>
<tr>
<td>S → robot var T T</td>
<td>begin 40 60 L robot bob T T ; move bob east 6 ; halt</td>
</tr>
<tr>
<td>T → int</td>
<td>begin 40 60 L robot bob 5 10 ; move bob east 6 ; halt</td>
</tr>
<tr>
<td>L → S ;</td>
<td>begin 40 60 S ; robot bob 5 10 ; move bob east 6 ; halt</td>
</tr>
<tr>
<td>S → obstacle T T</td>
<td>begin 40 60 obstacle T T ; robot bob 5 10 ; move bob east 6 ; halt</td>
</tr>
<tr>
<td>T → int</td>
<td>begin 40 60 obstacle T 11 ; robot bob 5 10 ; move bob east 6 ; halt</td>
</tr>
<tr>
<td>T → int</td>
<td>begin 40 60 obstacle 7 11 ; robot bob 5 10 ; move bob east 6 ; halt</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 ROBOGO program.

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

\[ T \]

\[ 7 \]

In this case, the pointer to the node in the symbol table containing 7 is pushed on the STstack.

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

\[ \rightarrow 7 \]
\[ \rightarrow 60 \]
\[ \rightarrow 40 \]
T → int

In this case, the pointer to the node in the symbol table containing 11 is pushed on the STstack.

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

→ 11
→ 7
→ 60
→ 40

S → obstacle T T

In this case, a node of type obst 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?

→ obst (which points to 7 and 11)
→ 60
→ 40
\[ L \rightarrow S ; \]

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

\[ S \rightarrow \text{robot} \text{ var } T \text{ T} \]
$L \rightarrow L S$ ;

$D \rightarrow \text{east}$ and $T \rightarrow \text{int}$
\[
S \rightarrow \text{move var D T}
\]

\[
L \rightarrow L S ;
\]
Part 4 - Execution of ROBOGO programs

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

In the example above, one would traverse the syntax tree and output JAWAA commands to a .anim file to 1) create an initial room of size 40 by 60, 2) create an obstacle at position (7,11), 3) create a robot named bob at position (5,10), 4) show the robot bob moving (cell by cell) 6 places to the east.

INPUT:
The input is a ROBOGO program. You may assume the tokens for ROBOGO 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 ROBOGO program is syntactically correct or not. If it is syntactically correct, then run the ROBOGO program and produce a .anim file. You’ll also need to create a .html file (just once). To run the JAWAA animation, load the .html page, adjust the speed control bar, click “start”, and you should seen your animation. If the ROBOGO program is not syntactically correct, your program should send a text message to the .anim file indicating this. That is, when you click start a message such as “Not syntactically correct” should be displayed.

You might want to setup your program to write to an .anim file in your public_html directory. Then each time you run your program, it will automatically write to this file and you will just have to reload the browser page.
If the robot crashes into an obstacle, stop at that point. See the project 1 handout for a sample picture. The sample picture uses squares for robots and circles for obstacles. Feel free to come up with your own representation, but make sure it is well documented at the top of your program.

**THE PROGRAM AND ITS SUBMISSION**

Your program should be written C++ and compile on the acpub machines. (Use the g++ compilers). You must also use the Makefile given in `~rodger/cps140/robogo`

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 a class or classes. Comments should describe each part of the class(es).

- **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!

**SUBMIT PART 1:**

If your userid is `abc`, then name your robogo program `abc.robogo` and submit it by typing:

```
~rodger/bin/submit140 robogo abc.robogo
```

**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.

Submit your program by using the submit140 command. For example, suppose you have a makefile called `Makefile`, a C++ program called `project3.cpp` and `project3.h`. To send these files, type

```
~rodger/bin/submit140 prog3 README Makefile project3.cpp project3.h
```

where `prog3` is the assignment name. This command should work on the CS machines and 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.

EXTRA CREDIT (3 pts)
You must turn the extra credit in at the same time with your program.

- Display (flash) an error message in the animation if an object (obstacle or robot) has coordinates outside of the window.
- Display (flash) 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.
- If a robot crashes into an obstacle, show fireworks!