The purpose of this assignment is to write an interpreter for the AZBOT programming language (see the project 1 and project 2 handouts for a description of the tokens and the grammar of the AZBOT programming language). Your program will read in a data file containing an AZBOT program, and if it is a syntactically correct AZBOT program, then you will interpret the program and graphically indicate robots, obstacles, and robot movement.

DESCRIPTION OF YOUR PROGRAM

Given a sample AZBOT 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 AZBOT 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. You can remove the output statements from this part.

Part 3 - The Syntax Tree

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

Whenever structure is recognized in an AZBOT 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 a reference to “bob” in the symbol table, to a node containing the direction “east” (created earlier) 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 a reference to the two syntax trees.

In order to keep track of the syntax trees, a stack called STstack will contain a reference 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 references on the STstack are
popped and joined together in a new syntax tree. Then the reference 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 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 nonterminal’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. When an AZBOT 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.

Types of nodes for syntax trees:

- **begin** - begin i j <\texttt{list}> halt - This type of node represents the beginning of a AZBOT program and has four parts. The first part tells the type of the node, begin, the second and third parts are references to the integers i and j in the symbol table, and the fourth part is a reference 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 reference to v in the symbol table, and the third and fourth parts are references 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 references 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 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.
• *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 references 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 reference 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 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).

• *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 reference to a syntax tree that represents the body of the do statement. The third and fourth parts are references 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 AZBOT program.

```
*-- program 1
begin 40 60
    obstacle 7 11 ;
    robot bob 5 10 ;
    move bob east 6 ;
halt
```

This AZBOT 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 T 10 ; 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 AZBOT program.

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

\[
\begin{array}{c}
T \\
\Downarrow \\
7
\end{array}
\]

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

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

\[
\begin{array}{c}
\rightarrow 7 \\
\rightarrow 60 \\
\rightarrow 40
\end{array}
\]

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

\[
\begin{array}{c}
T \\
\Downarrow \\
T \\
\Downarrow \\
7 \\
\Downarrow \\
11
\end{array}
\]

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

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

\[
\begin{array}{c}
\rightarrow 11 \\
\rightarrow 7 \\
\rightarrow 60 \\
\rightarrow 40
\end{array}
\]

\[ S \rightarrow \text{obstacle } T \ T \]

\[
\begin{array}{c}
S \\
\Downarrow \\
obst
\Downarrow \\
\rightarrow 7 \\
\rightarrow 11
\end{array}
\]

In this case, a node of type \textit{obst} 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?

→ obst (which points to 7 and 11)
→ 60
→ 40

L → S ;

T → int, then T → int

S → robot var T T
\( L \rightarrow L S ; \)

\[
\begin{array}{c}
L \\
\text{seq} \\
\text{obst} \quad \text{robot} \\
7 \quad 11 \quad \text{bob} \quad 5 \quad 10
\end{array}
\]

\( D \rightarrow \text{east} \quad \text{and} \quad T \rightarrow \text{int} \)

\[
\begin{array}{c}
L \\
\text{seq} \\
\text{obst} \quad \text{robot} \\
7 \quad 11 \quad \text{bob} \quad 5 \quad 10
\end{array}
\]

\( D \quad T \\
\text{east} \quad 6 \)
$S \rightarrow \text{move var D T}$

$L \rightarrow L S ;$

$L \rightarrow L S ;$

$L \rightarrow L S ;$
Part 4 - Execution of AZBOT programs

If the parser identifies that the AZBOT program is syntactically correct, then one can walk through the syntax tree and "run" the AZBOT 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 the program of robots moving in a room. One possibility is to generate an animation such as 1) an initial room of size 40 by 60, 2) an obstacle at position (7,11), 3) 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 AZBOT program. You may assume the tokens for AZBOT programs are all valid. The format of the data file is the same as it was in projects 1 and 2.

Note that you will still have to read in the parsedata file to build the parse table. Also, your program should still prompt the user for the name of the input file and then read from that file. This will make it easier to test your program on several data files.

OUTPUT:

Indicate whether the AZBOT program is syntactically correct or not. If it is syntactically correct, run the AZBOT program and produce a graphical simulation of the AZBOT program. If the AZBOT program is not syntactically correct, your program should display a text message indicating this, such as “Not syntactically correct”. If the robot crashes into an obstacle, stop at that point.

For the animation, you will produce JSAWAA commands into a .txt file that you can then cut and paste into the code window on the JSAWAA web page. Then click on Run Code and you can watch your animation. (see below for more info on JSAWAA).
See the project 1 handout for a sample picture. This picture is not done with JSAWAA. 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.

**EXTRA CREDIT (3 pts)**
Enhance the animation with by showing the following in the JSAWAA animation.

- Display an error message in the animation if an object (obstacle or robot) has coordinates outside of the window, or runs off the edge of the canvas.
- Display 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!

You must turn the extra credit in at the same time with your program.

**THE PROGRAM AND ITS SUBMISSION**
**REQUIREMENTS:**

- Your program should be written in Java.
- You should start by making a copy of your program from project 2.
- The name of the file with main should be called `project3.java`
- Your program should prompt the user for the name of the AZBOT program to test.
- Your program should output JSAWAA commands that can run in the JSAWAA web page. It is located here: https://www2.cs.duke.edu/csed/jsawaa/ There are two sample files. The one called sample.txt shows how to create and move simple objects such as circle, rectangle, line and text, and shows how to make a grid. This is likely all you need to use to do this assignment. The second file has additional JSAWAA commands but some of them are still being tested, use those at your own risk! Documentation for JSAWAA and the two sample files are on the assignment page with this assignment.
- Submit your program using websubmit under `project3`. A link to websubmit is on the assignment page. You can submit a .zip file if you have multiple files.
- In addition to submitting your program, you must fill out the REFLECT form on the assignment page.

**GRADING**
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 multiple methods and/or classes. Comments should describe each part of the methods/class(es).
• **Liberal use of comments** - In addition to the comment for each module, each nontrivial section of code 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 part of your grade is based on meeting the specifications of the assignment. If you do not get your program correctly running, for small amount of partial credit you may generate output that identifies which part of your program are correctly working. This output must also be clearly understandable or no credit will be given!

**Late Policy**

Programs not submitted by the due date are penalized 10% up to three days late.

**No LATE projects accepted after Thursday, April 25, 11:59pm.**

You must meet with Prof. Rodger if your program is not turned in one week after the deadline.