No LATE projects accepted after Thursday, April 24, 11:59pm.

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

DESCRIPTION OF YOUR PROGRAM

Given a sample GOOGBOT 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 GOOGBOT 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 GOOGBOT program, you will construct a syntax tree that represents the semantics of the GOOGBOT program. The tree can be built as the GOOGBOT program is parsed.

Whenever structure is recognized in a GOOGBOT 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 a GOOGBOT 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 <list> halt* - This type of node represents the beginning of a GOOGBOT 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 GOOGBOT program.

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

This GOOGBOT 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 GOOGBOT program.

T → int

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

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?

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

T → int

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

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?

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

S → obstacle T T

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

In this case, a node of type 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
\( L \rightarrow LS; \)

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

\[
\begin{align*}
\text{seq} & \quad \text{obst} & \quad \text{robot} \\
7 & \quad 11 & \quad \text{bob} & \quad 5 & \quad 10 \\
\end{align*}
\]
$S \rightarrow \text{move var D T}$

$L$

```
seq

obst
7 11

robot
bob 5 10

move
bob east 6
```

$L \rightarrow L S ;$

$L$

```

seq

seq

obst
7 11

robot
bob 5 10

move
bob east 6
```
Part 4 - Execution of GOOGBOT programs

If the parser identifies that the GOOGBOT program is syntactically correct, then one can walk through the syntax tree and “run” the GOOGBOT 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 GOOGBOT program. You may assume the tokens for GOOGBOT 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 GOOGBOT program is syntactically correct or not. If it is syntactically correct, then run the GOOGBOT program and produce a graphical simulation of the GOOGBOT program. If the GOOGBOT 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. 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.

**Graphics**
The focus of this assignment is to learn about how parsing works. Graphics added are an extra bells and whistle.

AT THE MINIMUM, you should store the grid in a 2d array with different symbols representing robots, the paths of robots, obstacles, etc. Then print out all error messages and the 2d array after the simulation is complete. (note you must indicate the path the robots are moved with appropriate symbols). Each symbol used and what it represents should be described in your README file.

**EXTRA CREDIT (5 pts)**

EXTRA CREDIT: Create an animation of the program with simple graphical objects for the robots and obstacles such as blue circle, red square, etc. Here errors should be displayed in the graphics. In addition:

- 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!

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

**THE PROGRAM**

Your program should be written in Java 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 classes. Comments should be included for each method to explain the purpose of the method.

- **Liberal use of comments** - In addition to the comment for each part of a class, 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 names** - Give appropriate names that describe their function for variables, methods, and classes.

- **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 partial credit you may generate output that identifies which part of your program is correctly working. This output must also be clearly understandable or no credit will be given!
SUBMIT

You should create a file called README that contains your name, the amount of time the project took, and anyone you received help from.

Also include instructions for running your program, to help the grader in running it.

Submit your project using Eclipse and Ambient under project3.

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

Programs that are up to 2 days late (weekends do not count as late days) will be 10% off. Programs that are up to 5 days late are 20%.