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

The purpose of this assignment is to write an interpreter for the ROBOBOT programming language (see the project 1 and project 2 handouts for a description of the tokens and the grammar of the ROBOBOT programming language). Your program will read in a data file containing a ROBOBOT program, and if it is a syntactically correct ROBOBOT program, then you will interpret the program and graphically indicate robots, obstacles, and robot movement. The graphics can be done either in Java or you can use the scripting language JAWAA. If using JAWAA, 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 ROBOBOT 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 ROBOBOT 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. You can remove the output statements from this part.

Part 3 - The Syntax Tree

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

Whenever structure is recognized in a ROBOBOT 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 ROBOBOT 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 ROBOBOT 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 ROBOBOT program.

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

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

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

\[ \underline{T} \]

\[ \downarrow \]

\[ 7 \]

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?

\[
\begin{align*}
\rightarrow 7 \\
\rightarrow 60 \\
\rightarrow 40
\end{align*}
\]

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

\[ \underline{T} \quad \underline{T} \]

\[ \downarrow \quad \downarrow \]

\[ 7 \quad 11 \]

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?
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 → L S ;

D → east and T → int
S \rightarrow \text{move} \ \text{var} \ D \ T

L \rightarrow L \ S ;

L \rightarrow L \ S ;
Part 4 - Execution of ROBOBOT programs

If the parser identifies that the ROBOBOT program is syntactically correct, then one can walk through the syntax tree and "run" the ROBOBOT 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 graphic commands to generate 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 ROBOBOT program. You may assume the tokens for ROBOBOT 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 ROBOBOT program is syntactically correct or not. If it is syntactically correct, then run the ROBOBOT program and produce a graphical simulation of the ROBOBOT program. If the ROBOBOT 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.

THE PROGRAM

Your program should be written in Java and use Eclipse. For the graphics, you can use either Java or JAWAA (described at the end of the handout).

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!

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

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**

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!

**JAWAA - Getting Started**

You can use either Java graphics or JAWAA to create the animation from your program. This describes how to use JAWAA. You are on your own if you want to use graphics with Java.

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 cs 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.cs.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 example**

Consider the following ROBOBOT program.

```
/* program 1 */
begin 40 60
  obstacle 7 11;
  robot bob 5 10;
  move bob east 6;
halt
```
Here might be the corresponding JAWAA code. Note that everything has been magnified by 10 so you can see the robots and obstacles.

```
rectangle r1 0 0 400 600 black lightgray

text t1 500 50 "Example Blown up to see objects" black 12
delay 1000

changeParam t1 text "Obstacle"
rectangle r2 70 110 10 10 red red
delay 1000

changeParam t1 text "robot"
circle bob 50 100 10 blue blue
delay 1000

changeParam t1 text "robot moves"
moveRelative bob 60 0
```

You can see this animation in robobaby1.html and robobaby1.anim on the CompSci 140 assignment page.

**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 `robobaby2.anim`.

To run this program, copy the `robobaby2.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 `robobaby2.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. If you have problems viewing changes, it may be a refresh browser problem. Try copying the `.anim` and `.html` to a new name robobaby2.html and robobaby2.anim and that will force it to refresh.
- JAWAA may not work in all browsers. Try another browser if you cannot get it to work.

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

```
www.cs.duke.edu/csed/jawaa2
```