No LATE projects accepted after Wednesday, April 26, 11:59pm.

The purpose of this assignment is to write an interpreter for the MOVEShapes programming language (see the project 1 and project 2 handouts for a description of the tokens and the grammar of the MOVEShapes programming language). Your program will read in a data file containing a MOVEShapes program, and if it is a syntactically correct MOVEShapes program, then you will interpret the program and draw and moves shapes in a window.

DESCRIPTION OF YOUR PROGRAM

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

Whenever structure is recognized in a MOVEShapes 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 references to “bob” in the symbol table, to 10 in the symbol table and to “skip” in the symbol table.

```
move
    bob
    10
    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 references to the two syntax trees.

In order to keep track of the syntax trees, a stack called STstack will contain references 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 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 reference on the STstack that points to the terminal in the symbol table), the top two references on the STstack are popped and joined together in a new syntax tree node. Then the reference to this new syntax tree is placed on the stack. 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 terminal’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. 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 MOVEShapes 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. When a lookahead that is also in the symbol table (such as a variable) is pushed onto the parsing stack, then a reference 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 MOVEShapes program and has three parts. The first part tells the type of the node, *program*, the second part is a reference to a size node, and the third part is a reference 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 references 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 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.

• *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 reference to *v* in the symbol table, the third through sixth parts are references to integers or variables in the symbol table, and the last part is a reference 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 reference to *v* in the symbol table, the third through sixth parts are references to integers or variables in the symbol table, and the last part is a reference 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 references 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 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).

• *for v = a to b by c do <stmts>* - This type of node has six parts. The first part identifies the node as a *for* node. The second part is a reference to variable in the symbol table. The third through fifth parts are references to either variables or integers in the symbol table. The sixth part is reference 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 MOVEShapes program.

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

This MOVEShapes 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 MOVEShapes program.
Z → size int int

In this case, the two references to 300 and 200 in the symbol table are on the STstack. A size node is created, the references to 300 and 200 are popped and put into the size node, and a reference to this size node is pushed on the STstack.

T → int

NOTE: What does the STstack look like at this point?
In this case, the reference to the symbol table for 5 is already on the STstack. So is the reference 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 reference to 5 off the stack and then push it back on, or just do nothing.

→ 5
→ skip
→ size 300 200
S → var = T

In this case, a node of type `asgn` 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?

→ 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: $T \rightarrow \text{int}$

```
T → int
size
300 200
Z
asgn
5skip
L T T T T
10 20 40 70
```

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

$\rightarrow$ 70
$\rightarrow$ 40
$\rightarrow$ 20
$\rightarrow$ 10
$\rightarrow$ fred
$\rightarrow$ asgn (which points to skip and 5)
$\rightarrow$ size 300 200

```
S → rect var T T T T c
Z
rect
fred
20
40
70
S
```

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

$\rightarrow$ rect node
$\rightarrow$ asgn node
$\rightarrow$ size 300 200
L → L S ;

\| size
| 300 200

T → var and T → int

\| size
| 300 200

L → L S ;
\[ S \rightarrow \text{move var T T} \]

\[ L \rightarrow L \ S \ ; \]

\[ Z \]

size \[\begin{array}{c}
300 \\
200 
\end{array}\]

move \[\begin{array}{c}
\text{fred} \\
\text{skip 0}
\end{array}\]

seq \[\begin{array}{c}
\text{rect} \\
\text{fred 10} \\
\text{red 20 40 70}
\end{array}\]

asgn \[\begin{array}{c}
\text{skip 5}
\end{array}\]

S \[\begin{array}{c}
\text{move}
\end{array}\]

L \[\begin{array}{c}
\text{seq}
\end{array}\]

Z \[\begin{array}{c}
\text{size}
\end{array}\]

move \[\begin{array}{c}
\text{fred} \\
\text{skip 0}
\end{array}\]

seq \[\begin{array}{c}
\text{rect} \\
\text{fred 10} \\
\text{red 20 40 70}
\end{array}\]

asgn \[\begin{array}{c}
\text{skip 5}
\end{array}\]
Part 4 - Execution of MOVEShapes programs

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

How to do the animation?

You can choose how to do the animation. Here are some choices.

1. You can use Python Turtle graphics if you are using Python.

2. You can use the tool JSAWAA that is a scripting language. Your program outputs JSAWAA commands and then you cut and paste them into a window that will interpret the animation.

3. Any other way you know how to do animation.
INPUT:
The input is a MOVEShapes program. You may assume the tokens for MOVEShapes 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:
If the MOVEShapes program is syntactically correct, then run the MOVEShapes program and produce suitable output for an animation.

THE PROGRAM AND ITS SUBMISSION
REQUIREMENTS:

• Your program should be written in Java or Python.

• You should start by making a copy of your program from project 2.

• If written in Java, the name of the file with main should be called Project3.java. If written in Python, the name of the file with main should be called Project3.py.

• Your program should prompt the user for the name of the PROGBOT program to test. If that file is in the same folder as your code, then your program should run on that file.

• Submit your program as a .zip file under Project3Files in Sakai under assignments.

• You must also submit a video that is 15 minutes or less where you explain and show us the modules/methods you wrote and what each one does, and run your program on at least three examples. The video can be submitted one day late with no penalty. Submit the video in Sakai under Project3Video OR submit a link to the video (if you submit a link, make sure we have access to view the video).

You can make the video however you want. One way is to use zoom.

• If you want to use JSAWAA, 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.

Documentation for JSAWAA and the two sample files are on the assignment page with this assignment.

• 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!

**No LATE projects accepted after Wednesday, April 26, 11:59pm.**

**EXTRA CREDIT: Error Handling (3 pts)**

The extra credit must be turned in with your program 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 in your animation, and then remove the message, or make the message float off the screen by moving it far off to the right.

- (2 pts) Print an error message if an object (line or rectangle) has coordinates outside of the 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 your animation to indicate that an error occurred.