No LATE projects accepted after Friday, May 1, 8am.

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

http://www.cs.duke.edu/~wcp/jawaa.html

DESCRIPTION OF YOUR PROGRAM

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

Whenever structure is recognized in a CAMO 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 bobo 4” is reduced to “Statement”, a syntax tree can represent the fact that the cat bobo should move 4 spaces in the current direction. We will create a node of type “cat”. This node should contain a pointer to “bobo” in the symbol table.

```
move

bobo

4
```

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.
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 CAMO 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:

- **size** - *size i j begin* <list> *halt* - This type of node represents the beginning of a CAMO program and has four parts. The first part tells the type of the node, *size*, 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.

- **cat** *v i j d* - This type of node has five parts. The first part tells the type of the node, *cat*, the second part is a pointer to *v* in the symbol table, the third and fourth parts are pointers to *i* and *j* in the symbol table, and the fifth part *d* is the direction.

- **mouse** *v i j d* - This type of node is a mouse node with five parts and is similar to the cat node.

- **hole** *i j* - This type of node has three parts. The first part tells the type of the node, *hole*, and the second and third parts are pointers to *i* and *j* 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 pointers to syntax trees, where those statements in the second part’s syntax tree should be executed before those statements in the third part’s syntax tree.
• *move v i* - This type of node has three 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 part points to the integer i in the symbol table.

• *clockwise v* - This type of node has two parts. The first part tells the type of the node, *clockwise*, and the second part points to the variable v in the symbol table.

• *repeat i stmts end* - This type of node has three parts. The first part identifies the node as a *repeat* node, The second part is a pointer to the integer i in the symbol table, and the third part is a pointer to a syntax tree that represents the body of the do statement. The meaning of the repeat statement is to execute the statements in the body i times.

Consider the following CAMO program.

```plaintext
// program 1
size 30 40
begin
    cat char 20 21 east ;
    move char 4 ;
halt
```

This CAMO 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 → size int int begin L halt</td>
<td>size 30 40 begin L halt</td>
</tr>
<tr>
<td>L → L S ;</td>
<td>size 30 40 begin L S ; halt</td>
</tr>
<tr>
<td>S → move var int</td>
<td>size 30 40 begin L move char 4 ; halt</td>
</tr>
<tr>
<td>L → S ;</td>
<td>size 30 40 begin S ; move char 4 ; halt</td>
</tr>
<tr>
<td>S → cat var int int D</td>
<td>size 30 40 begin cat char 20 21 D ; move char 4 ; halt</td>
</tr>
<tr>
<td>D → east</td>
<td>size 30 40 begin cat char 20 21 east ; move char 4 ; 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 CAMO program.

D → *east*

```
The meaning of the repeat statement is to execute the statements in the body i times.

In this case, the pointer to the node in the symbol table containing east is pushed on the STstack.
Here char, 20 and 21 would have already been shifted onto the STstack, and the direction node D is already on the STstack pointing to east. These 4 items are popped off the STstack, a cat node is made and the cat node is pushed onto the STstack.

This is interpreted as the top of the STstack is popped and then pushed, or you can do nothing since the stack doesn’t change in this case.

The char and 4 have already been pushed onto the STstack. They are popped off, a move node is created and pushed onto the STstack.
The top two items are popped off the STstack and joined together in a sequence node, which is then pushed on the STstack.

\[ P \rightarrow \text{size} \ \text{int} \ \text{int} \ \text{begin} \ \text{L} \ \text{halt} \]

The two ints have been shifted onto the top of the STstack. Three items are popped off the stack and joined together in a size node, and then pushed back onto the stack, where it should be the only thing on the stack.

**Part 4 - Execution of CAMO programs**

If the parser identifies that the CAMO program is syntactically correct, then one can walk through the syntax tree and "run" the CAMO 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 1) use jawaa to create an initial window of size 30 by 40 2) create a cat at position (20, 21) headed in the direction east, 3) show the cat moving (cell by cell) 4 positions to the east.

**INPUT:**

The input is a CAMO program. You may assume the tokens for CAMO 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 CAMO program is syntactically correct or not. If it is syntactically correct, then run the CAMO program and produce suitable output for JAWAA.
THE PROGRAM AND ITS SUBMISSION

Your program should be written in C++ and compile on the acpub or CS machines. (Use the g++ compiler).

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 modules. Each module should have a single purpose that is described in a block comment at the beginning of the module.

- **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!

You should create a file called README that contains your name, the amount of time the project took, and anyone you received help from. If you have multiple files, then you should use a Makefile. Submit your program by using the submit140 command. For example, suppose you have a makefile called `Makefile`, a C++ program called `project3.cc` and `project3.h`. To send these files, type

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

where prog3 is the assignment name. This command works only 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.

No late programs will be accepted after Friday, May 1 8am!

**EXTRA CREDIT** (1 pt) Create a interesting camo program. Submit it with your program. You can also submit it early and I will put this on a web page.

**EXTRA CREDIT** (4 pts)

For extra credit, your program should follow the rules listed below, generating error messages in the animation when rules aren’t followed and if multiple items are allowed on the same position, displaying the appropriate critter for that position.
RULES for Cat and Mice

- A mouse can hide in a hole, so display the hole if there is a mouse on the same position.

- Cats can walk over holes, but not go in them, so in this case display the cat until the cat is gone, then display the hole again.

- If a cat and mouse are on the same square (the mouse is not in a hole), then display the cat and assume the cat has eaten the mouse. You can remove the mouse. Any other statements that refer to the mouse are invalid, and an error message should flash in the animation.

- Two cats cannot be on the same square. This is an error. Remove the cat that entered the position last and keep the program running. It is then an error to refer to the cat that was removed. Flash an error message if this happens.

- Multiple mice can be on the same square if the square is a hole. Multiple mice cannot be on the same square otherwise. In that case, remove the mouse that entered the position last and flash error messages for any later statements that refer to that mouse.

- Display (flash) an error message in theJava window if an object (cat, mouse or hole) has coordinates outside of the window.