An elementary school would like to have a simple animation program to teach kids a little bit about computer science, in particular how programming works. Many of the kids know nothing about computer science, so a programming language made simple for animation (PLaMS) is needed for them. They’ve hired you to write an interpreter for PLaMS defined below.

The PLaMS programming language has a program definition (shown first) and five types of statements:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>size ( i \ j \ [ \ stmts ] )</td>
<td>program definition - defines size of animation window (width ( i ), height ( j ))</td>
</tr>
<tr>
<td>( v = a ; )</td>
<td>an assignment statement</td>
</tr>
<tr>
<td>line ( v \ a \ b \ c \ d \ color ; )</td>
<td>draw a line named ( v ) from point ((a, b)) to point ((c, d))</td>
</tr>
<tr>
<td>rect ( v \ a \ b \ c \ d \ color ; )</td>
<td>draw a rectangle named ( v ) with top left corner at point ((a, b)), width ( c ) and height ( d )</td>
</tr>
<tr>
<td>move ( v \ a \ b ; )</td>
<td>move ( v ) ( a ) spaces in the x-direction and ( b ) spaces in the y-direction</td>
</tr>
<tr>
<td>for ( v = b \ to \ c \ by \ d \ do stmts \end ; )</td>
<td>assign ( v ) the value ( b ), ((<em>)) execute ( stmts ), add ( d ) to ( v ), if ( v \leq c ) then repeat starting at ((</em>))</td>
</tr>
</tbody>
</table>

where \( v \) is a variable, \( a, b, c, \) and \( d \) are either variables or integers, \( i \) and \( j \) are integers, \( color \) is one of 8 valid colors, and \( stmts \) represent 1 or more valid statements.

Here is a sample PLaMS program that draws 2 rectangles and a line between them in the animation window. Then the rectangles and line move, with the big rectangle and line moving to the right and the small rectangle moving down and out of the animation window. A \% indicates a comment, meaning that whatever follows to the end of the line is a comment.

```plaintext
size 200 200 \[
rect fred 10 20 50 60 red ;
rect josh 30 70 10 20 blue ;
line sarah 10 20 30 70 green ;
for x = 1 to 10 by 1 do
    move fred 5 0 ;
    move sarah 5 0 ;
    move josh 5 20 ;
end ;
\]
```
The animation resulting from this program would look initially look like:

After the animation completes, it would look like (note the big square is the window, the little rectangle has moved outside the window):

The interpreter will be built in three parts (projects 1, 2 and 3). In this project, you will write a scanner that will identify the elementary parts (tokens) of a PLaMS program and store these parts for later use. In project 2, you will write a parser that will identify syntactically correct PLaMS programs. Project 3 will further extend the parser into an interpreter that will execute a PLaMS program and draw the resulting picture.

**DESCRIPTION OF THE SCANNER**

Given a sample PLaMS program, your first task is to identify all its parts (or tokens). Your program should include a scanner and a driver.

The purpose of the scanner is to find the next token in your program, enter its value into a symbol table that handles searches and insertions, and return 1) a reference to the tokens location in the data structure, and 2) a unique symbol, called the *token type*, which indicates the type of the token. Not every token is entered into the symbol table, but for those that are, make sure that there is only one copy of each. Thus, upon encountering a token, search
the symbol table first to see if it is already there, and if not add it to the symbol table. In either case you need to return a reference to the token.

The purpose of the driver is to repeatedly call the scanner requesting the next token type and the pointer to its value. For this project, the driver will print the token type and its value when it receives them and then request the next token (thus losing the information about the previous token). Although we are throwing away this information now, we will use it in project 2.

All the input is handled in the scanner, not in the driver. The first time the scanner is called, read in the first line of the PLaMS program and store it in a buffer. Scan the buffer until a token is identified, process the token and return. The next time the scanner is called, continue processing the buffer in the position right after where the last token was found. Whenever the end of the buffer is reached, the scanner should read in another line of input.

**TOK EN S**

The tokens of a PLaMS program consist of keywords, variable names, colors, integer constants, and punctuation symbols. Tokens are separated by blanks, end-of-line, and end-of-file.

Not all tokens are entered into the symbol table. If they are to be entered, then a string value and an integer value are inserted for each token.

The tokens of the PLaMS programming language and their associated types are:

**Keywords:** Keywords are not entered into the symbol table. They have no value. For each keyword found, return its type and NULL for its value. The uppercase and lowercase of the same letter should be treated as the same, so **FOr** is the same as for.

<table>
<thead>
<tr>
<th>KEYWORD</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>s</td>
</tr>
<tr>
<td>move</td>
<td>m</td>
</tr>
<tr>
<td>line</td>
<td>l</td>
</tr>
<tr>
<td>rect</td>
<td>r</td>
</tr>
<tr>
<td>for</td>
<td>f</td>
</tr>
<tr>
<td>to</td>
<td>t</td>
</tr>
<tr>
<td>by</td>
<td>b</td>
</tr>
<tr>
<td>do</td>
<td>d</td>
</tr>
<tr>
<td>end</td>
<td>e</td>
</tr>
</tbody>
</table>

**Variables:** Variables are entered into the symbol table. Valid variable names may contain 1-8 lowercase or uppercase letters. The uppercase and lowercase of the same letter should be treated as the same, so **SUm** is the same variable as sum. The type of a variable is v. The string value associated with a variable is the name of the variable, with all uppercase letters converted to lowercase. Its integer value is set to 0 for now (it is not needed until project 3). (Exceptions: Keywords are not variables. So, **for** is a keyword, not a variable!)

**Colors:** Colors are entered into the symbol table. The type of a color is c and the string value is one of 8 possible colors: red, blue, yellow, green, cyan, orange, black and pink. The integer value is 0 and will not be needed.
**Integers:** Integers are entered into the symbol table. Valid integers may contain 1-8 digits (0-9). If it starts with 0 then it must be of length 1. Integers can be positive or negative. The type of an integer is \( i \). Integers are read in as strings. Store the string value and convert the string to an integer, and also store the integer value (it will not be used until project 3).

**Punctuation Symbols:** These are not entered into the symbol table. They do not have values. For each symbol found, return its type and NULL for its value.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>[</td>
<td>]</td>
</tr>
</tbody>
</table>

Note that the minus sign that is part of the integer is not a punctuation symbol.

**Comments are not tokens!** In addition to tokens, your program may contain comments. A comment begins with \% and continues to the end of the line. All comments are to be ignored. When a comment is encountered, ignore the rest of the line, and start searching on the next line for the next token. There is no type associated with a comment.

**INPUT**

A data file consists of one PLaMS program. You may assume that PLaMS programs contain valid tokens. To ensure your program runs correctly, you should create your own data files for testing.

A sample data file is:

```
% program 1 %
size 300 300 [
  top = 200 ;
  line ray 75 Top 75 100 red ;
  For x = 5 to 35 by 5 do % move line 7 times
    move ray x 5 ;
  end ;
]
```

**OUTPUT**

For each PLaMS program print out the following information for each token in three columns: the type of token, the string value, and the integer value. If the token is not entered into the symbol table, then the string and integer values are left blank.

Possible output for the sample data file above might be:
Error Handling

You should handle files that contain invalid tokens. When an invalid token is found, report it as an error including the line number where the error occurred, and continue processing. For example, consider the following PLaMS program.
This program is loaded with syntactic errors, however, for project 1 identify only invalid tokens. The syntactic errors will be caught in project 2.

The invalid tokens above are:

- In line 1: twentyfive (too long)
- In line 2: end; (there is no separator between “end” and “;”, so they are treated as one token, which is invalid.)
- In line 3: * is an invalid token (note everything to the right of the % is a comment)
- In line 4: #B and ! are invalid

When an invalid token is encountered in the scanner, print an error message and the token, then continue scanning for the next token.

THE PROGRAM

Your program should be written in Java 5 or C++ 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 method, 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!

**SUBMISSION**

Include a README file that describes how to run your program. Your program should work for several data files, so you may want to prompt the user to type in the name of the data file.

Submit your program with Eclipse and Ambient under CompSci 140 **project1**.

Programs should be submitted by midnight (23:59:59) on the due date.

**LATE PENALTIES**

See the syllabus on the web for late penalties.

**EXTRA CREDIT (4 pts)**

For extra credit, if a word is not a valid token, assume that it contains tokens that are not separated by whitespace, try to identify the tokens, print a warning message, and then return the tokens one at a time as valid tokens. If a variable name is in front of a keyword, just treat this as an invalid variable name.

For any part that cannot be identified, report an error, discard the invalid token, and check the next token. Do not return an invalid token to the driver.

**Examples:**

The word **one15**; is actually three tokens: one, 15, and ;. Return “one” to the driver. The next time the driver calls the scanner, return “15”. The next time the driver calls the scanner, return “;”. You should either print one warning message for all three tokens, or three separate warning messages.

The word **add6tosome** is three valid tokens: add, 6, and tosome.

The word **twentyfive** should be reported as an error, variable name too long.

The word **begin*** should be reported as the keyword begin and an invalid token *.

The words **starmove** and **movestar** are valid variable names. Even though movestar contains the keyword move, this is a valid variable name, so don’t assume that an error has been made.

The words **starburstmove** and **starmoveburst** should be reported as errors, variable name too long. These contain the keyword move, but you are not required to detect keywords concatenated with variables.