The company AMAZOOGLE has hired you to write an interpreter for a simple programming language called AZBOT for programming their robots. This language allows the programmer to specify the location of obstacles and the starting position and movements of robots. For valid AZBOT programs, you will visualize the motion of the robot in order to determine if it is going to crash into any obstacles.

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin ( i ) ( j ) ( stmts ) halt</td>
<td>program definition - defines height ( i ) and width ( j ) of the room.</td>
</tr>
<tr>
<td>obstacle ( a ) ( b ) ;</td>
<td>draw an obstacle at position ((a, b))</td>
</tr>
<tr>
<td>robot ( v ) ( a ) ( b ) ;</td>
<td>draw a robot with name ( v ) at position ((a, b))</td>
</tr>
<tr>
<td>add ( a ) to ( v ) ;</td>
<td>add statement</td>
</tr>
<tr>
<td>move ( v ) ( d ) ( a ) ;</td>
<td>move the robot ( v ), ( a ) spaces in direction ( d )</td>
</tr>
<tr>
<td>( v = a ) ;</td>
<td>an assignment statement</td>
</tr>
<tr>
<td>do ( stmts ) until ( a &gt; b ) ;</td>
<td>Execute ( stmts ), if ( a \leq b ) then repeat</td>
</tr>
</tbody>
</table>

where \( v \) is a variable, \( a \) and \( b \) are either variables or integers, \( i \) and \( j \) are integers, \( d \) is a direction (north, south, east or west) and \( stmts \) represents 1 or more valid statements.

Here is a sample AZBOT program that draws a few obstacles and then moves the robot through the room. We will assume that all rooms are a grid of points \((x,y)\) with width \( w \) and height \( h \) with \( x \) from 0 to width and \( y \) positions from 0 to height. The upper left point is point \((0,0)\). A comment starts with \(*-\) and continues to the end of a line, so the end of line marker is the end of a comment.

```
begin 80 100
    obstacle 8 11 ;
    obstacle 9 2 ;
    obstacle 5 6 ;
    robot rob 3 2 ;
    wall = 13 ;
    j = 1 ;
    do
        obstacle wall j ;
        add 1 to j ;
    until j > 10 ;
run = 4 ;
```

\(--\) room has height 80 and width 100 --*
\(--\) an obstacle at (8,11) --*
\(--\) an obstacle at (9,2) --*
\(--\) an obstacle at (5,6) --*
\(--\) robot rob starts at (3,2) --*
\(--\) a wall of obstacles --*
\(--\) move robot --*
move rob east run ;
move rob south 11 ;
move rob east run ;
move rob north 6 ;
move rob west 7 ;
halt

A picture showing the robot rob and obstacles from this program would look like:

In the picture, single obstacles (denoted as circles) are drawn in positions (8,11), (9,2), (5,6), and a wall of obstacles from (13,1) to (13,10). The robot rob starts in position (3,2) (denoted by the s in the square) and the robots movements are indicated by the squares. In this example, the robot safely avoided all the obstacles.

The interpreter for AZBOT will be built in three parts. For this project, you will write a scanner that will identify the elementary parts (tokens) of a AZBOT program and store these parts for later use. In project 2, you will write a parser that will identify syntactically correct AZBOT programs. Project 3 will further extend the parser into an interpreter that will execute a AZBOT program and generate an animation of obstacles and a moving robot.

**DESCRIPTION OF THE SCANNER**

Given a sample AZBOT program, your first task is to identify all its parts (or tokens).

The purpose of the scanner is to find the next token in your program, enter its value into a data structure (called a symbol table) that handles searches and insertions, and return 1) a reference to the tokens location in the symbol table, 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. If so, return a reference to its location. If it is not in the symbol table, insert it, and then return the reference to its location.
Your program will repeatedly compute the next token type and the reference to its location in the symbol table. For this project, you will print the token type and its values in the symbol table, 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 projects 2 and 3.

Be sure to keep track of the program line number for reporting where errors occur.

**TOKENS**

The tokens of a AZBOT program consist of keywords, variable names, 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 character value and an integer value are inserted for each token.

The tokens of the AZBOT 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. Keywords are only formed using lowercase letters. The uppercase and lowercase of the same letter should be treated as the same, so **beGIN** is the same as **begin**.

<table>
<thead>
<tr>
<th>KEYWORD</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>b</td>
</tr>
<tr>
<td>halt</td>
<td>h</td>
</tr>
<tr>
<td>obstacle</td>
<td>o</td>
</tr>
<tr>
<td>add</td>
<td>a</td>
</tr>
<tr>
<td>to</td>
<td>t</td>
</tr>
<tr>
<td>move</td>
<td>m</td>
</tr>
<tr>
<td>north</td>
<td>n</td>
</tr>
<tr>
<td>south</td>
<td>s</td>
</tr>
<tr>
<td>east</td>
<td>e</td>
</tr>
<tr>
<td>west</td>
<td>w</td>
</tr>
<tr>
<td>robot</td>
<td>r</td>
</tr>
<tr>
<td>do</td>
<td>d</td>
</tr>
<tr>
<td>until</td>
<td>u</td>
</tr>
</tbody>
</table>

**Variables:** Variables are entered into the symbol table. Valid variable names may contain 1-8 lowercase 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 character 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, **east** is a keyword, not a variable!)

**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. The type of an integer is **i**. Integers are read in as character strings. Store the character value and convert the character 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.
Comments are not tokens! In addition to tokens, your program may contain comments. A comment begins with \texttt{*=} and extends to the end of the line. All comments are to be ignored. When a comment is encountered, ignore everything to the end of the line. Since comments are not tokens, there is no type associated with a comment.

INPUT

A data file consists of one AZBOT program. An AZBOT program should end with the \texttt{.azb} extension. For example, a file might be named \texttt{p1.azb}. Sample data files will be available soon. These are not necessarily the data files that your program will be tested on. To ensure your program runs correctly, you should also create your own data files for testing. A sample data file is:

\begin{verbatim}
*--  program 1
begin 60 80
   obstacle 7 11 ;
   robot biff 6 8 ;
   skip = 6 ;
   move biff east skip ;
halt
\end{verbatim}
OUTPUT

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

Possible output for the sample data file above might be:

OUTPUT FOR PROGRAM

<table>
<thead>
<tr>
<th>TYPE</th>
<th>CH VALUE</th>
<th>INT VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>i</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>o</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>biff</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>i</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>skip</td>
<td>0</td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>biff</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>skip</td>
<td>0</td>
</tr>
<tr>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ERROR HANDLING

Your program should handle files that contain invalid tokens. When an invalid token is found, report it as an error and the line number it occurs on and continue processing.

For example, consider the following AZBOT program.

```
one twentyfive 15 ;
begin ; test end; ok
paint * rob B twentysix ;
move #B east -* x
```

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 and twentysix is too long
- In line 4: #B and -* are invalid tokens.

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

THE PROGRAM AND ITS SUBMISSION

REQUIREMENTS:

- Your program should be written in Java.
- The name of the file with main should be called project1.java
- Your program should prompt the user for the name of the AZBOT program to test.
- Submit your program using websubmit under project1. A link to websubmit is on the assignment page.
- 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!

**Late Policy**

Programs not submitted by the due date are penalized 10% up to three days late and 20% if four or more days late (Sunday does not count as a late day). You must meet with Prof. Rodger if your program is not turned in one week after the deadline.

**EXTRA CREDIT (3 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.

For any part that cannot be identified, report an error, discard the invalid token, and check the next token.

**Examples:**

The word `one15;` is actually three tokens: one, 15, and ;. Return “one” as a token, then return “15”, and then return “;”. You should either print one warning message for all three tokens, or three separate warning messages.

The word `add6tosum` is three valid tokens: add, 6, and tosum.

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.