**Collaboration Policy** You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own independently written solution. List the names of all of your collaborators. Do not seek published solutions for any assignments. If you use any published resources when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff.

**Objectives:**

1. Program a turing machine
2. See the similarity between automata and real-world computer architectures
3. Implement a parser for a turing machine's instruction set
4. Implement a simulator for a turing machine's instruction set on an input tape

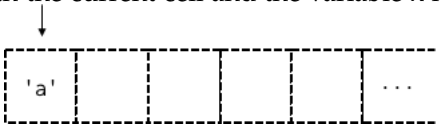
**Background** A turing machine is a finite state automata that can read from and write to an infinitely long input tape. It has a read head that can be moved along the tape, and can transition based symbols it reads from a cell. Based on this description, we have created an instruction set that directly maps to a turing machine's operations.

A `move <offset>` instruction shifts the read head on the input tape by the declared integer offset. For example, `move -2` shifts the read head to the left by two cells.

A `var <name>` instruction declares a variable with an associated name. A variable can store any printable ASCII character, either based on its integer value or character literal. The `read <name>` instruction stores the value held the tape's current cell into the variable with the associated name. To set a value on the tape, use the `write <operand>` instruction, which stores either an integer, character literal, or variable's value into the current cell. For example, after executing

```
var x
write 'a'
read x
```

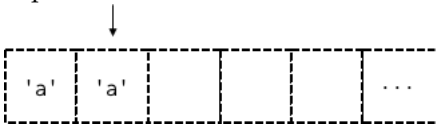
both the current cell and the variable x hold the character 'a':




But after executing the following,

```
move 1
write x
```

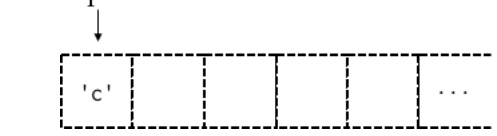
the tape would then hold



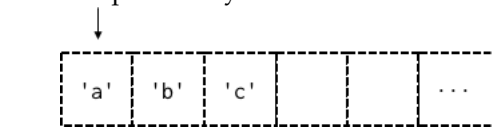

It is possible to jump to any labeled line in the program. Simply declare a line to be a label with a `<name>:`, and use the `jump <name>` instruction to reposition the program counter to that line. For instance, after executing the following

```
write 'c'
jump end
write 'a'
end:
```

the tape would hold



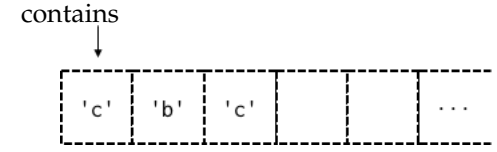

The machine implements conditional branching with the `skipif <condition>` instruction. This instruction skips over the next line if the given condition holds true. The condition is a boolean expression that compares the equality of two operands. For example, if the tape initially contains




and the following program is executed,

```
var x
read x
skipif x == 'a'
jump L
write 'c'
L:
```

the jump instruction is skipped over, because the value of x is 'a'. The resulting tape contains




A comment can be included on its own preceded with two forward slashes //. It is ignored during simulation.

The formal grammar for this instruction is given below. Note that it is context-free.

```
SourceFile = InstructionList .
InstructionList = { Instruction "\n" } [ Instruction ] .
Instruction =
   Label ":" |
   "var" Identifier |
   "skipif" BooleanExpr |
   "move" Integer |
   "jump" Label |
   "read" Identifier |
   "write" Operand |
   Comment |
    .
Label = Identifier .
Identifier = letter { letter | digit } .
letter = "a"  "z" | "A"  "Z" .
digit = "0"  "9" .
```

```
BooleanExpr = Operand ( "==" | "!=" ) Operand .
Operand = Identifier | Integer | "'" character "'" .
Integer = [ "+" | "-" ] digit { digit } .
Comment = "//" text .
```

**Part 1 (40 points)** For this part, you must implement the following programs using the instruction set above. Test your programs by using the online simulator located at tmsim.akhil.cc. Write each program in a file named `<program-name>.tm` for your submission.

1. `increment.tm`: Write a program that takes as input a binary number in reverse, and increments it by one. For instance, the decimal number `6` would correspond to the input string `011`, and the incremented result would be `111`. Example test cases:

| Input | Output |
|---|---|
| 0 | 1 |
| 01 | 11 |
| 1 | 01 |
| 11 | 001 |
| 101 | 111 |

2. `palindrome.tm`: Write a program that is a decider for palindromes. That is, for $\sum = \{\texttt{'a'}, \texttt{'b'}, \ldots, \texttt{'z'}\}$, $L = \{w \in \sum^* \mid w \text{ is a palindrome}\}$. The result should be a tape only containing a `1` in the first cell if the decider accepts, and `0` in the first cell if the decider rejects. Example test cases:

| Input | Output |
|---|---|
| | 1 |
| a | 1 |
| aa | 1 |
| ab | 0 |
| anna | 1 |
| madam | 1 |
| garage | 0 |

**Note: The program listed must match the file name! This is used for automated grading purposes!**

**Part 2 (30 points)** In this part, you will implement a parser that takes in a program written in the instruction set above as a string, and returns a `List<Instruction>`. The required class declarations are located in **Parser.java**. The `Instruction` class is already provided for you in **Instruction.java**. You are not required to handle comments or empty lines. Additionally, you can assume that all of the tokens on a line are delimited by a single space. Run **TestParser.java** to run tests against the parser. Only valid programs will be tested against the parser.

**Part 3 (30 points)** Here, you will implement a simulator that takes in a `List<Instruction>` and input tape, and returns the tape after executing the above instruction set. You are not required to handle invalid programs as all test programs are valid and halt. Run **TestSimulator.java** to run tests against the simulator.

*Implementation notes*: Empty cells can be treated as whitespace characters. If a `read` or `write` instruction attempts to execute on input array of zero-length, grow the array to contain a whitespace character. Additionally, if a `move` instruction attempts to move past the end of the input array, grow the array to the appropriate size and initialize the remaining characters as whitespace.

**Submission** When you are ready for submission, run **Submit.java** in the working directory containing all of your source files. This will produce a zip file that you should upload to collab.