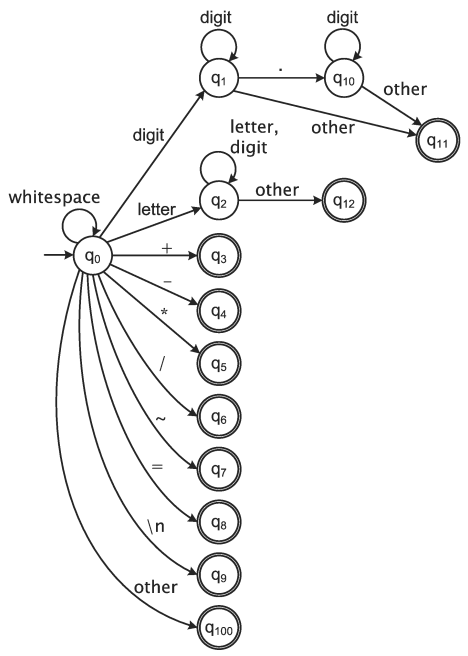
CS 327
Spring 2022
Programming Assignment 4

Introduction

For this assignment you will use a deterministic finite state automaton and push-down automaton to create the essential portions of a desk calculator. The DFA will be used to tokenize or lexically analyze the input (that is, convert it from a stream of characters to a stream of tokens), and the PDA will evaluate the input. The rest of the program will be provided to you as a Java file.

Lexical Analysis

As noted, lexical analysis converts a stream of characters into a stream of tokens. A **lexer** or **tokenizer** does this by scanning the input one character at a time, and using this as the input to a DFA that halts when it reaches a final state. The final state indicates the token—the token codes are defined as constants in the `Lexer` class. The diagram below shows the DFA for recognizing tokens of the desk calculator. Note that digits are what one would expect, letters include upper and lower case letters plus the underscore character, and whitespace is spaces and tabs.



As this DFA shows, the tokens of the desk calculator are numbers (recognized in state q_{11}), variables (recognized in state q_{12}), the end-of-line (q_9), the binary operators +, -, *, /, and = (states q_3, q_4, q_5, q_6andq_8), and the unary operator ~ (numeric negation, or minus, recognized in state q_7).

The token recognized in state q_{100} is the bad token (some unrecognized character, at least in that context).

One slight complication is that numbers and variables are only recognized when either whitespace or the first character of some other token appears. In this case the other character must be returned to the input stream. There is a method to do this in the code provided to you.

You will need to implement this DFA. There are three standard ways to implement DFAs.

- Keep track of the current state in an integer state variable and, in a loop, move from state to state on input characters according to the transition function represented as a two dimensional array indexed by states and characters. This table-driven approach is fast but does no allow for any additional processing in the states.This is an option for your implementation.
- Keep track of the current state in an integer state variables and move from state to state by switching on the current state, and for each case, either switching on the current character, or using conditionals on the current character to determine the next state. This code-driven approach makes it easy to add extra processing to each state if it is needed. This is an option for your implementation.
- Make each state a class that implements a `State` interface, and keep track of the current state in a `State` state variable. In a loop, move from state to state by asking the current state what its next state is, given a character. This class-based approach allows for all sorts of very fancy processing in each state (where of course arbitrary fields and methods may reside). This is overkill for the the current project, so it is not a good option for your implementation.

In either of the first two alternatives, one nice trick is to make final states negative. Then you can tell easily when to stop execution (when the state is less that 0), but you still know which state you ended up in (the negation of the final state variable value).

Your tokenizer must also accumulate the text of the current token, which is needed for numbers and variables and also comes in handy for error messages. The `Lexer` class has a text field for accumulating the characters in the current token, and a `getToken()` method that clients can use to obtain it.

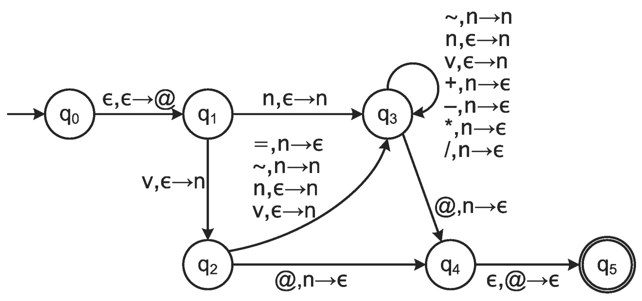
Evaluation

Evaluation piggy-backs on parsing; **parsing** is recognizing whether a stream of tokens is a string in a language. The language of our desk calculator is defined by the following grammar.

<line> → (<assign> | <exp>) <eol>
<assign> → <variable> = <exp>
<exp> → <exp> <unary-op> | <exp> <exp> <binary-op> | <variable> | <number>
<unary-op> → ~
<binary-op> → + | - | * | /

The non-terminals <eol>, <variable>, and <number> are of course the tokens end-of-line, variable, and number. The rest of the terminal symbols you should recognize as the other tokens in the language.

This is a context-free language so we can parse it with a PDA. The algorithm we discussed for this is non-deterministic. However, this particular language can be parsed with a simple deterministic PDA, shown in the figure below.



In this PDA, the symbol v stands for a variable, the symbol n stands for a number, and @ stands for the end-of-line token.

You must implement this PDA to recognize (and evaluate) lines of desk calculator input. Note that you don't actually need states q_0 and q_5 because they are only there to manage the bottom-of-stack marker, and you can tell when the stack is empty without that. Hence you only need to implement the machine without these states, using q_1 as the initial state and q_4 as the final state.

You can implement this PDA in the same way that you can implement a DFA (that is, table-driven, code-driven, or using classes), with the addition of a stack. However, a table-driven approach is less appropriate because it is awkward to manipulate the stack. Also, you will need to add code for doing arithmetic and other evaluation tasks to each state. The class-based approach is still overkill for this problem, so that pretty much means you should use a code-driven approach, switching on the current state and then switching or otherwise deciding what to do based on the next input token and the status of the stack.

As tokens are processed by the PDA, note that when a variable is encountered, a number should be pushed on the stack; this number is the value of the variable. Similarly, when a number is encountered in the input it is pushed on the stack. If a ~ token is processed, there must be a number on the stack, and a number is pushed on the stack. In this case the top number is popped from the stack and its negation is pushed on. In the case of a binary operator, there must be a number on the stack and it is removed. Of course what really needs to happen for evaluation to take place is that *two* numbers are popped from the stack, the operation is performed (with the first number number popped as the right argument and the second number popped as the left argument), and the result pushed on the stack. And so forth. So you need to figure out what processing occurs in each state of the PDA recognizer to evaluate an expression.

Only numbers go on the stack. It starts empty at the beginning of a line, and at the end of a line there should only one number in it. If the stack becomes empty prematurely, then there are not enough operands for the operators, and if there is more than one value in the stack at the end of the line, then there are too many operands for the operators. Also, if a bad token appears, that is an error. You should detect and report all these errors. There is an `error()` method provided that you can call with an error message. It will report the error and flush the input (go to the next line). Your PDA should quit on an error and go on to the next line if any of these things happen.

There are variables in this language. You can assign to them, and if you don't make an assignment, the value of the current line is assigned to the variable "it" by default. How do you manage these variables? Quite simply, use a hash map from variable names to values (this is called a **symbol table**). When you encounter a variable, get its value from or set its value in the hash map, as appropriate. The default value of a variable that has not yet been assigned one is 0.0 (this way there is never a problem with an undefined variable).

Your PDA implementation must be in the `evaluate()` method in the `Evaluator` class. All numbers are `Double` objects, so that is what should be stored on the stack. Note that Java's boxing and unboxing features make it easy to do arithmetic with `Double` objects. The `evaluate()` method must return the one and only `Double` value in the stack a the end of line processing if all goes well; otherwise (that is, if there is an error) it must return `null`.

Exiting the Program

There are two ways to exit the program: when the end of the input is reached (because the user typed CTRL-D or the end of a file redirected to standard input is encountered), or when the variable "exit" appears as the expression on a line (in other words, when the user just types "exit" on a line and hits return).

The code in the skeleton that gets the next character of input in the lexer already handles the end of input case. You will have to write code to handle the other case in the `evaluate()` method. Specifically, once a line is processed, `evaluate()` must check whether the "exit" variable appears as the expression on a line. If so, the method should print "Bye" on a line by itself and call `System.exit()` to halt the program.

Deliverables

The code skeleton that you must complete is in the file `Evaluator.java`. The lexer is implemented as an inner class just so that everything is in one file; a better design would be to have this class in its own file. You must finish the `Lexer` class by writing the `nextToken()` method and the `Evaluator` class by finishing the `evaluate()` method. Note that there are several methods already in these classes for you to use (in particular, character input, results output, error messages, and overall control); you may write other methods if you need them.

When you have finished writing the missing code, run `Evaluator` and you should have a desk calculator. You should write the lexer first and test it with JUnit, then finish `evaluate()` once the lexer works reliably.

Please submit the completed `Evaluator.java` file on Canvas.