CS 4321/5321 Project 1

Fall 2017

Due September 8th, 2017, at 11:59 pm

This project is out of 100 points and counts for 12% of your overall grade.

Before you start this assignment, make sure you have **read the course policies document (available in CMS) and completed the CMS quiz on the course policies. You must complete this quiz to pass the course.** We assume you are familiar with the course policies. Therefore, we don't repeat info about group work, late submissions, academic integrity etc in these instructions.

1 Goals and important points

Although this project in not directly related to database material yet, it is very important. It aims:

- to teach you (or remind you of) the overall structure of a compiler/interpreter. In the remainder of the course you will be developing a simple SQL interpreter. It is important to understand the basic architecture and abstractions on something simple like arithmetic expressions first.
- to provide a warm-up/review of essential data structures, design patterns and Java programming.

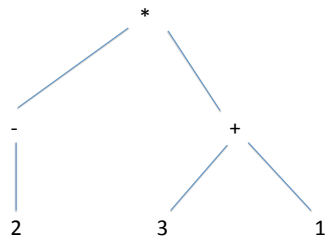
This project is intended to be very easy, even if you have no knowledge of compilers. It requires a CS 2110-level grasp of Java programming as well as data structures (trees, lists, stacks), a solid understanding of recursion and an ability to understand new abstractions. Our instructions are deliberately high-level; we will not always tell you exactly how to implement something.

Note that subsequent projects will be *much* harder than this one; Project 2 will be available in CMS soon so you can take a look at it for comparison. If you find Project 1 difficult and/or need a lot of help from the staff to finish it, **you need to drop the class**.

2 Overview

In this project, you will be implementing a simple interpreter for arithmetic expressions. Your code will take in a string that looks something like `"-2.0 * (3.0 + 1.0)"` and do various interesting things with it. These things include:

- *parsing* the string to create a tree representation of the expression - a tree for our example string is shown below.



- walking the tree you just created for a variety of purposes:
  - converting the tree back into a string, i.e. pretty-printing the expression
  - evaluating the expression to an actual number (in our case, -8)
  - converting the tree to a linked list which represents the same expression in prefix (Polish) or postfix (Reverse Polish) form. You will also write code to process the resulting list to evaluate the expression and to pretty-print it.

You need to be familiar with prefix and postfix notation; you can find extensive information on both online. For our example expression, a prefix representation is

$$* \sim 2.0 + 3.0\ 1.0$$

and a postfix representation is

$$2.0 \sim 3.0\ 1.0 + *$$

In prefix/postfix form, we will use the symbol $\sim$ to represent a unary minus operator, as it requires special handling to distinguish it from the binary minus operator. (Think about converting something like $(-2) - (-3)$ to postfix notation and evaluating the resulting expression and you will see the problem and the need for a separate symbol).

An expression in prefix or postfix form is easily represented as a linked list in the obvious way; for example our expression $* \sim 2.0 + 3.0\ 1.0$ can be represented as a linked list with six elements, the first containing $*$, the second $\sim$ and so on.

3 Architecture and code overview

Next we explain the overall structure of the code. At this point it is useful to look at the skeleton code we provided. Please open it and follow along while you read this.

Every compiler and interpreter can be (roughly) partitioned into a front-end and a back-end. A front-end is a component that takes in some input. The input is a single string in our case, but in a "real" compiler it would be a source code file or a set of such files. The front-end takes the input and translates it into some suitable intermediate representation, which is usually a tree. A back-end is a component that processes the tree to do something useful, e.g. to transform it into a different intermediate representation, to generate code in a different language, or to evaluate the code/expression represented by the tree.

3.1 Intermediate representations

In this project our main intermediate representation is a tree. Please look at the package `cs4321.project.tree`; you will see we have several different tree node types. For example, there is a node for addition, division, etc. Nodes can be leaf nodes (no children but contain a numerical value), unary nodes (one child) or binary nodes (two children). Our second intermediate representation is a singly-linked list of nodes, found in `cs4321.project.list`; we will use this to represent an expression in prefix or postfix form.

3.2 Front-end

In this project, the front-end is just one file: `Parser.java`. You will implement this as part of the project. It takes in a String and produces a tree in our intermediate representation.

3.3 Back-end

The back-end in our project has a bit more functionality. If you wish, you can think of it as having multiple different back-ends. Each component uses the visitor pattern (more on this below) and traverses the appropriate intermediate representation (tree or list) to do something useful.

- `TreeVisitor.java` and `ListVisitor.java` - these are interfaces for each of the backend files that do "real work".
- `PrintTreeVisitor.java` - this is the only backend class provided for you. It traverses a tree to create a String that represents a parenthesized, pretty-printed version of the expression in infix form.
- `EvaluateTreeVisitor.java` - you will implement this. This traverses a tree to evaluate the corresponding expression to a single number.
- `BuildPrefixExpressionTreeVisitor.java` - you will implement this. This traverses a tree to create a list representing the same expression in prefix form.
- `BuildPostfixExpressionTreeVisitor.java` - you will implement this. This traverses a tree to create a list representing the same expression in postfix form.
- `PrintListVisitor.java` - you will implement this. It traverses a list representing an expression in either prefix or postfix form, and creates a String that represents the expression.
- `EvaluatePrefixListVisitor.java` - you will implement this. It traverses a list representing an expression in prefix form, and evaluates the expression to a single number.
- `EvaluatePostfixListVisitor.java` - you will implement this. It traverses a list representing an expression in postfix form, and evaluates the expression to a single number.

From the above description, you are probably noticing that:

- we want to traverse the trees/lists in many different ways for many different purposes, and
- each tree/list can contain many different types of nodes.

At any point in time, the logic to be executed depends both on the type of traversal we want and the type of the node. For example, if we are processing a tree and we encounter a node, the work we do depends on the type of the node (we will do one thing for an addition node and another for a multiplication node). It also depends on what the traversal is doing - is it producing a string, or is it evaluating the tree to a number, or is it constructing a list?

The cleanest way to handle this with an abstraction is through the *visitor pattern*. Each of the traversal classes implements a `Visitor` interface that includes a `visit` method for each node type. For example, if you look at `TreeVisitor.java`, you will see it includes six different `visit` methods, one for each possible type of tree node. All of our backend traversal classes implement either `TreeVisitor` or `ListVisitor`.

Correspondingly, each node in our trees and in our lists has a `accept` method (take a look at e.g. `AdditionTreeNode`). This takes in a concrete visitor (e.g. a `PrintTreeVisitor`, or a `EvaluateTreeVisitor`), and makes a callback to the `visit` method in the visitor. This might seem circular, however, note that the callback passes in the node that is being visited. Therefore the actual `visit` method that is called is dependent on the type of the node being visited. If the visitor hits an `AdditionTreeNode`, it will execute different code from when it hits a `DivisionTreeNode`. For more information about the visitor pattern, please refer to the additional handout in CMS. It is very important that you become comfortable with the visitor pattern as it will be used extensively in later projects.

3.4 Unit tests

The `test` subfolder contains a number of files with some JUnit test cases for each of the front-end and back-end files. You will need to implement more test cases for this project. If you are unfamiliar with JUnit, you need to learn about it yourself; many resources are available online.

4 What you need to do

For this assignment, you have to do three main things:

- implement the front-end in `Parser.java`
- implement the back-end functionality in the six backend visitor files discussed in Section 3.3
- create unit tests for all your files.

You may find it easier to start with the back-end first, to familiarize you with the tree intermediate representation. At the very least, we would recommend starting with `EvaluateTreeVisitor.java` and completing that before attempting the front-end.

We recommend you write unit tests as you go rather than at the end. For grading, you need to **add at least two unit tests to each of the eight test files we have provided you**, and submit those. For thorough testing, you should write many, many more unit tests than just two per file. If you look at the unit test files we have provided, each file tests a single component that is obvious from the name of the file. The only exception is `FullSystemTest.java`, which includes tests that use multiple components and put them together to test end-to-end functionality. You should only run these tests at the very end, once you have implemented, tested and debugged each component separately.

4.1 Back-end

We give a few notes, pointers and hints on each of the back-end classes, in suggested implementation order. **None of your classes need to handle invalid inputs.**

4.1.1 `EvaluateTreeVisitor.java`

This evaluates the tree to a single number. It is probably best to use a stack for this and a postorder traversal of the tree. For a binary node, recursively evaluate its left subtree to a number and push the result onto the stack. Then evaluate its right subtree and also push that number to the stack. Finally, pop the top two values off the stack and do something appropriate based on the node you are processing. Push the result onto the stack. (You will need to adjust a bit to handle a unary node, but the general idea is the same). At the end, the stack should contain just a single number which is the result of the whole expression.

4.1.2 `PrintListVisitor.java`

This should be fairly easy to figure out – it will be similar to the provided `PrintTreeVisitor.java`, except that you need to traverse a list instead of a tree. It will also help you debug the lists you create in other visitor classes, so it's good to implement this one early. **When printing a unary minus, use the symbol $\sim$ rather than $-$.**

4.1.3 `BuildPrefixExpressionTreeVisitor.java` and `BuildPostfixExpressionTreeVisitor.java`

Now that you are familiar with our tree and list representations, this should not be too hard. You need to traverse the tree (you will need one kind of traversal for prefix expressions, another for postfix expressions) and build up a running list as you go. This is similar to the way you build up a running String in `PrintTreeVisitor.java`.

4.1.4 `EvaluatePostfixListVisitor.java`

Before you start on this, be sure you know how to get valid *inputs* for this visitor. That is, be sure you can construct a list corresponding to a valid postfix expression – either manually or using your tested and debugged `BuildPostfixExpressionTreeVisitor.java`.

The best way to proceed here is with a stack, using the standard postfix evaluation algorithm. Maintain a stack of operands as you traverse your list. If you encounter a number, push it on the stack. If you encounter an operand, pop an appropriate number of operands of the stack, evaluate the operator, and push the result onto the stack. At the end of evaluation, the stack should contain a single value: the result.

4.1.5 `EvaluatePrefixListVisitor.java`

There are several ways to evaluate a prefix expression; one of them is to read the expression right-to-left and run essentially the postfix algorithm described above. However, for this project, we want you to traverse the list *left-to-right*. If you traverse the list right-to-left (or reverse it), you will get zero points for this part.

The suggested algorithm is as follows. Maintain *two* stacks, one for operands and one for operators. The operand stack will contain numbers, the operator stack will contain pairs. Each pair consists of the operator, and the number of *remaining operands* which we still need to see for that operator.

We traverse the string. If we encounter an operator, we initialize an appropriate pair and push it on the operator stack. For instance if we are processing the expression `+ 3 4`, we encounter a `+` and push $(+, 2)$ on the operator stack. The 2 indicates we are still waiting for two more operands. When we encounter a number, we push it on the operand stack, and look at the operator stack. If we find an operator there, we decrement the number of operands remaining by 1. In our example with `+ 3 4`, after processing + and 3, the operand stack will contain 3 and the operator stack will contain the pair $(+, 1)$, to show that the plus is waiting for one more operand. If the number of operands remaining drops to zero, we are ready to evaluate the operator. This will happen in our example after we process the 4; now the operator stack contains $(+, 0)$ and the operand stack contains 3 and 4. We evaluate the operator by:

- popping the appropriate number of operands off the operand stack
- popping the pair corresponding to the operator off the operator stack
- evaluating the operator
- pushing the result onto the operand stack (note that this may require decrementing the remaining number of operands for the *next* operator on the operator stack)

Once this terminates and we have traversed the entire list, the operand stack should contain a single number representing the result of the expression evaluation.

4.2 Front-end

The front-end is just one class, `Parser.java`. We strongly recommend that you be very familiar with the tree intermediate representation before you start. If you have never seen a parser before, you may find this the most confusing part of the project. You can find additional information by searching about recursive descent parsing online, and in books such as "Compilers: Principles, Techniques and Tools" which is available on Uris Library Reserve.

4.2.1 Overview

The parser will transform a string into a tree representing the expression which the string encodes. First, we give some important information about the strings you are expected to handle.

- You will only be given valid strings to parse - a valid string contains only numbers that parse to a java double, $+$, $-$, $/$, $*$, (, and) symbols, correctly nested to represent a valid expression. All symbols will be separated by spaces. All parentheses will be properly nested and matched, but there may be extra parentheses. For example (((1.0 + 2.0))) is a valid input you should handle.
- The symbol $-$ may be used as either a unary operator or a binary operator, your parser needs to figure out which one is desired. All instances of $-$ should be parsed as either a unary or binary operator; for example on being fed $-$ 1.0 your parser should produce a tree consisting of a UnaryMinusTreeNode with a LeafTreeNode below it; the LeafTreeNode contains the value 1.0.

So, how do you go from a string to a tree? Parsing is a complex topic which is covered in depth in compilers classes. Here the goal is to give you a rough idea of how this works by writing the simplest possible parser - a *recursive descent parser*. In subsequent projects you will not be writing your own parser for SQL, but we do want you to understand at least one way to implement a parser.

At the core of every parser is a *grammar*. This is a set of rules that explains the structure of the intermediate representation (tree) you are trying to produce. We will use the following grammar:

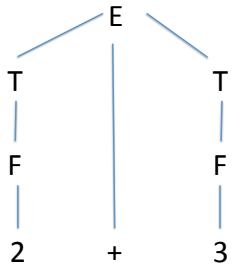
$$\texttt{E} := \texttt{T}\ \ \{+|-\ \ \texttt{T}\}^*$$
$$\texttt{T} := \texttt{F}\ \ \{*\ |\ /\ \texttt{F}\}^*$$
$$\texttt{F} := (\ \texttt{E}\ )\ \ |\ \ -\ \ \texttt{F}\ \ |\ \ \texttt{number}$$

The symbols :=, {, }, | and * (asterisk used as superscript) are only used in the notation of the grammar. The symbols `E`, `T` and `F` are called *nonterminals*; they stand for *expression*, *term* and *factor* respectively. These are not symbols that will appear in your input string, but they capture its structure in a fundamental way. The remaining symbols, namely +, −,∗, /, (,) and `number`, are called *terminals* and they will actually appear in your input (ok, `number` is not going to appear in your input, but it stands for any number which may appear in your input).

The top-level nonterminal is `E`. This means everything we parse will be an expression. Expressions can have several different forms. The first rule says that an expression can be a term, or a string of two or more terms separated by plus or minus signs. (In the grammar notation, enclosing something in { }* means "zero or more instances"). But what is a term? Well, the second rule says that a term is either a factor, or two or more factors separated by either times or division signs. What's a factor? The third rule says that a factor is either a number, or another factor with a minus sign in front of it, or an expression in parentheses.

The goal of parsing is to take in a string and use the rules above to "explain" why the string represents an expression `E`. As a simple example, consider the string "2 + 3" and try to parse it using our grammar. We are trying to show why it is an expression `E`. You see that by the third rule, 2 and 3 are both numbers, so they are both factors `F`. But, by the second rule, every factor `F` is also a term `T`, so 2 and 3 are both also terms. However, by the first rule, two terms separated by a plus sign are an expression, so the whole string does indeed represent an expression `E`.

A more formal way to think through the above example is to consider parsing as the process of building a *parse tree*. This tree will have nonterminals in internal nodes, and terminals at the leaves. At the root, we will always have the nonterminal `E`. The parse tree for our "2 + 3" string is shown below.



4.2.2 What you are going to implement

The parser you build will not actually construct the parse tree discussed above; rather, it will construct a tree that fits our tree intermediate representation. Nonetheless, these two trees are closely related.

If you look at `Parser.java`, note that you only have to implement three methods: `expression()`, `factor()` and `term()`. Each of these corresponds to one nonterminal and one rule in our grammar. The methods will, however, recursively call each other. For example, the `expression()` method will correspond to the grammar rule:

$$\texttt{E} := \texttt{T} \ \{+|- \ \texttt{T}\}^*$$

This rule has `T` on the right hand side, so the `expression()` method will call the `term()` method.

Note that the parser begins by taking in an input string and splitting it up into an array of *tokens*. It will traverse the array one token at a time and parse the string. At all times, there is a current token pointer which points to the next token to be processed; this pointer will be advanced as the tokens are processed/consumed.

The top level method is `parse()`. This just calls `expression()`, since we are trying to derive an expression. Now, if we are parsing an expression, we know it is a list of terms separated by + or - symbols. Therefore, we proceed as follows:

- call `term()` to parse the first term; this will consume the portion of the input string corresponding to that term. Don't worry about how `term()` works yet, we will come to that next. The call to `term()` will return a TreeNode; store it somewhere, say in `result`.
- while there are tokens left and the next token is + or -
  - consume the token corresponding to the + or -, i.e. advance the current token pointer.
  - parse the next term by calling `term()` again; store it in `result2`
  - connect `result` and `result2` as children of a new `AdditionTreeNode` or `SubtractionTreeNode` as appropriate
  - set `result` = the new TreeNode you just created
- now either there are no more tokens in the input or the next token is neither a + nor a -. Either way, we are done parsing the expression, so return `result`

The above is the pseudocode for `expression()`. Of course, you need to also implement the `term()` method. This will actually be very similar structurally to `expression()`, except that it will construct MultiplicationTreeNodes and DivisionTreeNodes rather than addition/subtraction nodes. `term()` will also call `factor()` as needed.

What about the `factor()` method? It needs to handle the grammar rule

$$\texttt{F} := (\ \texttt{E}\ ) \ | \ - \ \texttt{F} \ | \ \texttt{number}$$

Basically, it knows the next token is either a (, a - or a number. It will look at the next token and proceed accordingly. The easy case is when the next token is a number - it can just construct a LeafTreeNode with the number and return it. In the other two cases, it will need to make a call to either `factor()` (recursive call to itself), or to `expression()`, and do something with the result of that call. Note that parentheses are not featured anywhere in our tree representation, so in the case where a factor is a parenthesized expression, the `factor()` can just return the result of the `expression()` call it makes.

Note that the pseudocode above implies left-associativity of operators with the same precedence. Thus given the string 1.0 + 2.0 + 3.0 you should produce an AdditionTreeNode having as its left child another AdditionTreeNode corresponding to 1.0 + 2.0, and as its right child a LeafTreeNode for 3.0.

4.2.3 Suggested implementation strategy

This can be tricky to implement and debug if you try to do everything at once. We recommend starting with the simplest possible functionality - implementing the `factor()` method to handle the case where your whole expression is a single number. That is, start by implementing functionality to take a string like `"5.0"` and return a LeafTreeNode that contains 5.0. To achieve this, have `parse()` call `factor()` instead of `expression()` and implement just enough of `factor()` to produce a LeafTreeNode.

Once you have that working and tested, you can expand to the unary minus. Implement a second case in `factor()` to handle unary minus. Your goal should be to correctly parse strings like `"- 5.0"`, `"- - 5.0"`, and so on. Once that is working and tested, add the rest of the language features gradually.

5 Grading

We expect you to implement the project following the architecture we provided. We realize there are many other ways to implement the functionality of the project, most of them much simpler than what we are asking you to do. However, the whole point is to get you used to the visitor pattern, recursive tree/list traversals, and other concepts that are absolutely essential in the rest of the practicum.

With this in mind, here are rules you must follow. **If you disregard any of these rules, you may receive an arbitrarily large penalty to your score** even if you pass all our tests, and even though our normal grading scheme only allocates 10 points for code style (basically, our normal grading scheme becomes "null and void" if you disregard the below rules).

- Do not modify any of the code in the `cs4321.project1.tree` or `cs4321.project1.list` packages, or in `TreeVisitor.java`, `ListVisitor.java`, or `PrintTreeVisitor.java`.
- Do not remove any of the methods in any of the classes. You may add new methods if needed.
- You may add new classes in the `cs4321.project1` package - both inner classes and non-inner ones.
- You must implement the front-end logic we require using recursive descent parsing, and the back-end logic we require using the visitor pattern and recursive traversal of trees and lists. You may use a different algorithm from the ones we suggest in `EvaluateTreeVisitor.java`, `EvaluatePostfixListVisitor.java` and `EvaluatePrefixListVisitor.java`. However, any algorithm you use in those visitors must still walk the tree/list recursively node by node.
- In `EvaluatePrefixListVisitor.java` your algorithm must traverse the list left-to-right.

Exceptions to the above rules are only going to be granted by Lucja if you convince her that you understand the way the project is intended to be implemented and that you have a better/cleaner solution, rather than trying to get out of understanding/applying a key concept.

5.1 Code style and comments (10 points)

Provide comments for every method you implement. At minimum, the comment must include one sentence about the purpose/logic of the method, and `@params`/`@return` annotations for every argument/return value respectively. In addition, every class must have a comment describing the class and logic of any algorithm used in the class. Take a look at `PrintTreeVisitor.java` for an example of what we expect. In the `@author` annotation, include the names and netIDs of everyone who worked on the code. If you follow the above rules and don't write absolutely egregious code, you are likely to get the full 10 points for code style.

5.2 Unit tests (90 points)

Add at least two unit tests to each of the eight test files provided, for a total of 16 tests. Each test must come with a comment that specifies that this is one of the new tests you added, to make it easy for the grader to spot it.

Each unit test you provide is worth 1 points, for a total of 16 points (if you provide more than 16 tests, you don't get extra points). You can expect that if you write good-faith unit tests, all of your tests will receive 1 point. We will only deduct points for a unit test if:

- your own code does not pass your own unit test (yes, we will check this)
- you give something that is not a good-faith unit test. For example, if you just copy and paste one of the tests we provided, or write a test consisting of `assertEquals(1, 1);`.
- your test is not appropriate to the test file it is in. For example if your test is in the `PrintListVisitorTest.java` file but never creates a PrintListVisitor, rather, it tests something to do with evaluating expressions on trees, you will not get credit. Move that test to the appropriate file, and you will get credit.

For the remaining 74 points, we will run your code on our own unit tests. Passing the tests we provided with the skeleton code will give you 24 out of the 74 points.

6 Submission Instructions

Submit via CMS a .zip file containing:

- your Eclipse project folder
- optionally, a README file with anything you want the grader to know. If you added any .java files to the project rather than just modifying existing ones, note this down here and state which files are new. This file is also the place to note any known bugs, any idiosyncrasies of your code, etc.
- an `acknowledgments.txt` file if required under the academic integrity policy.