Project 2 Report: LISP Interpreter

Bryonna Klumker

LispInterMain is the **main** of this program. It is what asks what mode the user wants to use and is what captures input. If the input's parenthesis are balanced correctly (using the method in the **ParenthCheck** class which returns a boolean) **LispInterMain** sends the input to the **Parser**'s **parseMe** method.
The **Parser** class is the workhorse of this program. It takes a string, and shaves off its outmost parenthesis. Then it checks to see what the string contains to determine what course of evaluation must be taken. It's only method is the **parseMe** which is a recursive function that evaluates input.

Expression	Implementation
DEFINE a variable: '(define <var> <number>)'	● I created a the class **KeyVal** which just stores a value to a key. ● The **SymbolHelper** class which has an ArrayList (dynamic list object) that stores all KeyVal pairs. ● When "define" is in the input string the case is caught and parsed for <var> and <number> ● The SymbolHelper's addPair(String n, String m) will call a KeyVal constructor with n, m and add the new KeyVal to SymbolHelper's ArrayList. If <var> already exists as a key it just overwrites the old value with the new one.
VAR REFERENCE for any defined variable x: '(x)' will return the value	● When a variable is used the ArrayList is linearly searched by the SymbolHelper's GetVal method. ● If the variable is defined the value is returned as a string, if it is not defined the string "-NOVAL" is returned in order to throw a "not defined" error to the interpreter's interface.
SET!: '(set! <var> <exp>)' Evaluates the expression and sets <var> to the answer	● Works much the same as 'define' ● But parseMe(<exp>) is called to evaluate the mathematical expression so that <var> can be assigned to it.

COND: '(if (<cond>) (<Exp1>) (<Exp1>))' returns Exp1 if <cond> is True and Exp2 if <cond> is False	• The work for this is done in the Condition class • The string is parsed for parenthesis, context information is stored and used to parse out the conditional operator (<, >, =), the conditional operands, and the If Expression and the Else Expression. A switch statement is used for the conditional operator and parseMe is called on either the If or Else Expression, whatever one is appropriate for the test.
QUOTE: '(quote <string>)' like Linux's echo, just returns <string>	• Just splits the string on the space so that "quote" is at element 0, and adds all elements > 1 together with a space between them. Returns the constructed string minus "quote"
POLISH NOTATION MATH Prefix expression math. (Check README for more information	• The **PrefixEval** class's method evalExp() is called if the input string contains mathematical operators and/or one of the built-in functions. • The **PrefixEval**'s evalExp() has a copy of the **SymbolHelper**'s ArrayList (that the Parser maintains) so that it can use defined variables. • The input string is divided up into an array by spaces. Each element is parsed for a variable, a built-in op, a number, or an operator. • Two stacks are used in evalExp(), a number stack and an operator stack. The stacks are built by parsing each element of the input string that was split into a string array. Each element is parsed and added to the appropriate (numbers or operators) stack. • Every time the numbers stack has two numbers in it they are popped off, an operator is popped off, and using switch(operator) the correct operation is performed and the result is added to the numbers stack. The number of things on the stack leads to easily catching errors.
BUILT-IN FUNCTIONS: sin, cos, tan, sqrt	• This is done in **PrefixEval** by recognizing 'sin' 'cos' 'tan' and 'sqrt' and parsing out the parameter, and then calling the Math.java version of the function on the parameter.

DEFINE functions: ‘(lambda <name> (<params>) (<exp>))'	● Lambda functions are stored almost exactly like variables are. ● The **Lambda** class takes raw input and parses it for the name**,** the parameters, and the expression. Each parameter is stored as a character in a character ArrayList. The name and expression are stored as a strings. ● Every time a lambda function is defined a **Lambda** object is created for it and stored in the **LambdaHelper**’s ArrayList<Lambda>. ● If a defined lambda function is called, a temporary expression string is created by being set to the **Lambda**’s stored expression and then replacing the parameters in the call with the corresponding parameter in the **Lambda**’s parameter ArrayList, the numbers being assigned to the correct variable going off of the indexes of the character array.

2.) This project really solidified my understanding of syntax vs. semantics. All of the expressions had very similar syntax. I use the Java string method .contains() to evaluate which type of expression is being entered. I had to play around with the order of the tests in my code because of the way I handled recursion(if the expression minus the outer parentheses contains parentheses parseMe() is called again). So the **condition, set!,** and **lambda** cases had to be caught before the recursive condition, because they had similar syntax (embedded parenthesis) as the complex mathematical expressions. I realized just how hard it was to write for all the subtle differences in semantics and context and realized how difficult it is to create a robust interpreter that can handle every case.

It also solidified the my understanding of the OOP paradigm-- I needed a class for every little thing, and I realized that the whole computer is a System object to Java-- as well as the concept of automatic memory management (I, the user, had to do zero memory management because of Java’s garbage collector).

3.) These are the classes I haven't mentioned yet and a brief description of what they accomplish:
IsMathExp - Has a boolean method that checks if an expression contains mathematical operators or built in functions to determine whether it should be handled by **PrefixEval** or not.
ExpNode - This is the class that makes the recursion happen. If a mathematical expression is entered with nested parenthesis it is sent to be parsed and created as an **ExpNode** object, which has the parent expression and an ArrayList of the children expressions as attribute.

For example (+ + 6 (+ 2 3) (+ 2 2))
Will have a parent expression of "+ + 6"
And two child expressions "+ 2 3" and "+ 2 2"

If a child also has a child then an **ExpNode** will be created and stored instead of a simple string.
BuildExp - This class is also necessary for the recursion. It will take a newly created **ExpNode** and build the string by calling parseMe() on the children and adding the results to the parents. (it gets a copy of Parser in the method call so it has the variable lists and everything that has been entered so far). Then it adds the outer parenthesis so when the built string is returned to the Parser it can recursively call parseMe on the built string.

For example *the ExpNode created from:* (+ + 6 (+ 2 3) (+ 2 2))
Will become the built string sent back to the Parser: (+ + 6 5.0 4.0)

Then the built string is in the correct format for parseMe to be available to evaluate it immediately.

4. My interpreter works for every test case in my testcases.txt file but it could be better in the following ways:

- Allow for embedded expressions in conditionals: **Ex. >** (if (< 10 20) (+ + 10 20 (+ 10 20)) (- + 10 20 5)) *returns* 10 not 60.
- Allow for non-nested expressions to be at the end of a prefix expression. (I mentioned this can't be done in both the README and the '(help)' section of the program.) It doesn't crash the program but the output is as if the last number (the non-nested one) isn't in the expression.
- Allow for '(<number>)' input to output <number>. It does nothing currently.
- Allow for lambda defined functions to be used inside of mathematical expressions.

5. I feel like I'm always going to be bad at recognizing concepts that are more about categorizing things (like with principles of programming languages) than concrete concepts I can use in coding, but this project did a decent job of making me better at that. Besides that, I thought it was very fun and a refreshing challenge to build a program that has a useful purpose.

References:

- https://www.tutorialspoint.com
- http://stackoverflow.com/questions/1102891/how-to-check-if-a-string-is-numeric-in-java
- https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html
- http://stackoverflow.com/questions/7013590/how-to-use-replacechar-char-to-replace-all-instances-of-character-b-with-noth
- Louis Jencka and Nico (I'm sorry I don't know his last name), the SysAdmins, helped me figure out why my code wasn't working properly on the CS machines and how to fix it (it was just a line in the makefile to specify which version of java to use).