Project 2 Report

Nicholas Smith
New Mexico Institute of Mining and Technology
Department of Computer Science
801 Leroy Place
Socorro, New Mexico, 87801
nicholas.smith@student.nmt.edu

ABSTRACT

In this paper, the implementation of a basic LISP interpreter in Java is discussed. The LISP interpreter includes support for basic arithmetic, conditionals, user defined variables and functions, and several LISP functions such as quote.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Java, LISP

General Terms

Interpreter, Implementation

Keywords

LISP, Interpreter, Java.

1. INTRODUCTION

LISP is a simple functional programming language. Because it is simple, it is easy to write a basic interpreter for LISP in an Object Oriented Language. In this paper, I discuss my implementation of a LISP interpreter in the Object Oriented Language Java.

2. NEW INPUT: THE LEXER

When the user submits a new string for the Interpreter to evaluate, it first goes through a method in Interpreter called newInput(). This function creates a new Lexer object which parses the input string into a List of type String. This List contains only tokens relevant to the interpreter. Any spaces and unsupported characters, &, “, etc are ignored.

Overall the lexer is fairly simple. It iterates over each character in the input String. It compares each character to a set of character ranges, Letters, Atom, Literal, and Parens. If the character is a letter or numeric atom, it starts an inner loop to build a word or number. Otherwise it is added directly to the list of tokens. Once the word or number is completely built, it too is added to the list. The outer loop skips over any extra characters added. Once this is done, newInput() returns the result of calling the Interpreter method eval() with the tokens generated by the Lexer.

3. EVAL()

eval() is the method where the tokens produced by the Lexer are evaluated. This method is pretty much a task delegator with a few exceptions being the set!, quote, variable reference, and numeric atom cases, which are handled within eval() itself, implementation details to come.

Once eval() has checked that parentheses are balanced and that tokens is not null, eval() goes through each token passed to it and calls the appropriate function for the first operator it comes across, passing the parameters of the operation to the operation’s function as a sublist of the list tokens. eval() returns the result of this operation. Inside each operation’s method, the parameters are split into smaller sublists per parameter of the operation. Then the rules for this operation are applied to the result of calling eval() for each parameter and this result is returned.

This mutual recursion, methods which call each other until a base case is reached, allows the Interpreter to build a stack of operations it has to complete. Once all subsequent calls to eval() arre complete, the Interpreter can start evaluating the innermost operations to the outermost operations.

Finally, the result of the overall expression is returned.

The following subsections go over how each part of the interpreter was implemented with the exception of variables and functions which get their own sections later in the paper.

3.1 Arithmetic Operations

Included in this Interpreter are the built in arithmetic operations +, -, *, /, exp, and sqrt. Note that sin, cos, tan are not included. This is due to these functions returning decimal values and this interpreter being designed to work with integers only, even sqrt returns an integer. Due to this, I did not see the utility in implementing such functions.

With the exception of sqrt, which only takes one parameter, the other above-mentioned operations function almost the same. They each apply the following algorithm to obtain the parameters they require, then they apply their operation and return the result:

```
List<String> a = new ArrayList<String>();
List<String> b = new ArrayList<String>();
if(tokens.get(0).equals("(")) {
        a = tokens.subList(0,
indexOfClosingParen(tokens) + 1);

/*indexOfClosingParen returns the index of the
closing parenthesis of a List*/

} else {
        a.add(tokens.get(0));
}
if(tokens.get(a.size()).equals("(")) {
        b = tokens.subList(a.size(),
tokens.size());
} else {
        b.add(tokens.get(a.size()));
}

int aR = Integer.parseInt(eval(a, "(op)"));
int bR = Integer.parseInt(eval(b, "(op)"));
int result = aR (op) bR;

return "" + result;
```

This algorithm makes use of the indexes of the tokens List to separate out parameters a and b. If a and b are numbers/variables, eval(a/b) just returns their value, otherwise it conducts the operations held within each parameter and returns the respective results. Finally, the op is applied to aR and bR (a/b Result) and the result of this is returned.

sqrt functions much the same but only a List a is created as it only takes one parameter.

3.2 Boolean Comparison Operations

With the conditional statement being required, so were boolean operaterations. Included in this Interpreter are some basic comparisons: <, <=, >, >=, =, and !=. The &, |, and ! operations for chaining multiple comparisons are not included. The only real difference between these operations and the arithmetic ones is that result is of type boolean instead of type int.

3.3 Conditional Statements

A conditional takes three parameters, a condition, a statement to evaluate if the condition is true, and another statement if the condition is false. The same algorithm mentioned above is again applied, this time adapted for three parameters. Then the condition is evaluated. If the result of eval(cond, “if”) is true, then the condition true parameter is evaluated and returned. If false, the result of the condition false parameter is evaluated and returned.

3.4 Set!, Variable Reference, Quote and Numeric Atoms

If the token being evaluated is a variable reference or numeric atom, the value of that variable / atom is returned.

If the token is “quote” then the sublist following quote is returned without the closing parenthesis of quote. I.e. (quote (foo)) returns (foo).

If the token is “set” followed by a token “!” then the Interpreter searches its list of variables that have been initialized for the variable name which is located in tokens following “set” and “!”. After the variable is found, that variable has its value set to the evaluation of the tokens following the variable name.

3.5 Conclusion

Overall, evaluation is fairly simple, parse then call the appropriate operations which parse their arguments into the necessary parameters for the operation. These parameters are then evaluated and the results have the specified operation applied. The final result is then returned.

4. VARIABLES

Variable is a simple class included with the interpreter. It has two fields. One being its name, the other it's value (of type int). Upon the user calling (define name value), the String is tokenized and the define token is found. The eval() function then calls the define() function which applies the same algorithm as arithmetic/boolean operations to isolate the name and value of the variable. Then a new Variable object is created from the name and the result of evaluating the value expression. This object is stored in the Interpreter's local vars List.

The Variable class also includes a method to get the variable's value and one to set it which are used in Variable Reference and Set! respectively.

5. FUNCTIONS

The Function class contains three fields: fname (String), definition (List<String>), and params (List<String>). The Function class also contains a method which invokes(calls) the function.

5.1 Defun()

Upon finding a defun token, eval() calls the method defun(). In this function, the parameters necessary to create a new function, name, params, and a definition are parsed. Then, a new function, created from these parameters, is added to the Interpreters list of functions.

5.2 Function call

If the token being evaluated isn't any predefined operation, there are three choices left, function call, variable reference, or numeric atom. The Interpreter starts with function calls. By iterating over the list of Functions, the Interpreter can determine if the token is a function name. If it is then that function is "called" with the parameters following the function name in tokens. The function then builds a list to be evaluated from the definition and passed parameters. A detailed rundown of this process is included in the next section.

5.3 Parameter Agreement

In order to allow for complex function calls such as in the case:

(defun add (x y) (+ x y))
(add ((+ 3 4) 3))

, a condensing and expanding function were written within the Function class to force the parameters into agreement with the function's parameters. For example:

(add ((+ 3 4) 3)) tokenizes to:

[(, add, (, (, +, 3, 4,), 3,),)]

When add is called the passed params are:

[(, (, +, 3, 4,), 3,),)]

This is then condensed to:

[(, (+ 3 4), 3,)]

Then this is used to synthesize our definition with our parameters:

Def = [(, +, x, y)]

In our example:

x = (+ 3 4)
y = 3

/*Note: these are not actual assignments, just what our parameters are going to be interpreted as.*/

A List, toEval, is then built from the elements of def, substituting the defining parameters for the passed parameters:

toEval = [(, +, (+ 3 4), 3,)]

toEval is then expanded and evaluated:

[(, +, (, +, 3, 4,), 3)]

6. CONCLUSION

To implement the LISP interpreter, extensive use of the Java construct List from java.util.Collections was used. The use of List allowed for me to use the indices of the List to parse LISP expressions into their operators and their subsequent parameters. Once I figured out exactly how to do this, it was simple to apply operations to their parameters.

The next challenge was complex (non-atomic) parameters for user defined functions. Initially, I tried to directly substitute these complex parameters but that led to issues. So, I decided to try condensing the parameters before substituting them into the function's definition and then retokenizing the expression. This approach worked well and functions could be called by the user with complex parameters.

This project was an enjoyable test of my knowledge of Java. It required me to use everything from basic String methods to mutual recursion to creating a class capable of tokenizing a String. If I was to return to this project in the future, I would like to implement support for more data types including lists of data with various types as LISP is THE list processor.