Creating a LISP Interpreter Using Java

Myles Willis
Department of Computer Science
New Mexico Institute of Mining and Technology
Socorro, New Mexico, 87801
myles.willis@student.nmt.edu

ABSTRACT

This document will describe the process used to construct a LISP interpreter using Java along with each of the functions successfully implemented. The goal of the project is to create a program that will execute LISP commands using the syntax and semantics of the Common LISP programming Language. The following table show each LISP construct that has been completed successfully in the Java LISP interpreter

Lisp Construct	Status (Done, Partially done, not done)
Variable reference	Done
Constant literal	Done
Quotation	Done
Conditional	Done
Variable Definition	Done
Function call	Done
Assignment	Done
Function Definition	Not done
The arithmetic +, -, *, / operators on integer type.	Done
"car" and "cdr"	Done
The built-in function: "cons"	Done
sqrt, exp	Done
>, <, ==, <=, >=, !=	Done

CCS CONCEPTS

D.3.3 [**LISP**]: All

KEYWORDS

LISP. Interpreter, Java

1 INTRODUCTION

This text will provide an overview of the process used to create a LISP interpreter using Java. The following sections will describe each of the Java classes created, how they work, their purpose for the overall goal of making an interpreter, along with some details specific feature implementations.

2 JAVA CLASSES AND IMPLEMENTATIONS

2.1 Interpreter.Java

The Interpreter class acts as the front-end of the LISP interpreter tasked with collecting user input and providing system output for the evaluated LISP commands. The class uses a loop to prompt for user input and output results until the user enters the (quit) command. To parse the user input, a parser object is created then the input string is sent to its parseInput function (Section 2.2). Once the string is parsed a evaluator object is created and is sent the string of tokens to solve the user's LISP expression and return the answer (Section 2.5) Additionally, Interpreter.Java outputs the error messages from exceptions thrown by other functions in the interpreter and evaluator classes.

2.2 Parser.Java

After using Interpreter.Java to gather user-inputted expressions in String form, the string must be split into individual parts to allow for manipulation and interpretation. A Parse object consists of the user inputted string, a string array list to hold the individual parts of the input then a Token object to hold the input in the form of a list of Token. A parser object is created in the Interpreter class.

To prepare the input string for use as a list of strings in other functions, blank spaces are inserted around every opening and closing parenthesis. Once the spaces are inserted, the Java function split() is called on the string breaking each chunk of text into a single element to be placed in an arraylist of strings.

An additional function validateParenthesis is implemented to count the number of open and close parenthesis in user input. If the number of each parenthesis is not equal then the input has mis-matched parentheses and an error is thrown. For all user input that is valid, the created list of strings is sent to the parseTokenList() function (Section 2.4.1). Finally, the parser object's token parameter is set equal to the token returned by the parseTokenList() call.

2.3 **Operator.Java**

The Operator class is responsible for the definition of every implemented function used for evaluation of user inputted expressions. Each operator has their associated form (in the form of a string) along with type. There are arithmetic, boolean and lisp type operators. This class makes use of Java enumerations for its structure.

The arithmetic typed operators include: +, -, *, /, sqrt, and pow. Boolean operators include: <, <=, >=, =, !=, >, "and", "or, and "not". For LISP functions the lisp operators include: "car", "cdr", "cons", "if", "define" and "set!".

The Operator class also holds three functions: evalArithmetic, evalBool, and evalLisp. These functions are responsible for receiving values, performing a given operation, then returning the answer. The evalArithmetic function is able to perform the operations of the arithmetic operators on a list of doubles. EvalBool performs the boolean operators on a list of doubles and evalLisp performs the lisp operations on an arraylist of arguments.

2.*3.1 Implementing Arithmetic Operators, SQRT and POW LISP Construct*

The evalArithmetic function is used to implement the arithmetic operator LISP construct. The function is called on an operator object and includes a list of doubles as a parameter. Inside of the function there is a switch statement which has a case for each arithmetic operator. The ("+") case for example adds every element in the list together then returns a number token that contains the sum. In other cases such as ("-") or ("/") the first element in the included list is extracted as the divisor or minuend then those values are appropriately solved with the remaining arguments. Similarly, sqrt and pow have switch cases solved using the list of doubles along with conditional statements that ensure the functions have the appropriate number of arguments (sqrt requires a single number and pow requires two).

2.*3.2 Implementing Boolean Operators LISP Construct*

The boolean comparisons are handled the same way as the arithmetic operators in a function named evalBool. This function uses an array of doubles as a parameter. The switch statement holds cases for each boolean operator where the first element in the double list is compared to the rest of the list values. A boolean token is created equal to the result of the comparison.

2.*3.3 Implementing Lisp Operators LISP Constructs (car, cdr, cons, if)*

An additional function evalLisp uses a switch statement that has cases for each of the operator defined LISP functions. Cases include evaluation logic for car, cdr, cons and if. The evalLisp function has a parameter which requires a list of tokens. Conditional statements are used to check if the token list is properly formatted for each lisp operation. For example, "car" and "cdr" only require a single list to operate on, where "cons" requires two tokens (a atom/list as first token and a list as a second token) and "if" requires three tokens (a boolean token as first token, a literal token as the second and third token in list).

For the implementation for car, the first token in the array list of tokens is returned. When evaluating cdr, the first element of the list is removed, then a token list token is returned which is composed of the remaining list elements.To implement cons, the first token in the array list is added to the token list provided as the second element in the list. Finally, for the implementation of "if", the case returns the true case (second element of list) if the first list element is true or returns the false case (third element of list) if the first element is false. Each of these functions use the functions and structure of ArrayList objects in Java.

2.4 **Token.Java**

A crucial aspect of a LISP interpreter is correctly parsing and interpreting the input as LISP atoms, functions, lists and literals. The Token class' purpose is to facilitate the creation of each type of input type with their specific parameters and usages. Types of tokens include: number, boolean, operator, string, tokenlist tokens.

A token is designed to hold the necessary data or content of atoms, lists, operators, functions etc. A token is also designed to hold lists of other tokens (e.g An entire LISP expression is parsed as a single token list storing a series of tokens). Tokens have a boolean parameter "literal" which marks it to be a literal atom/list or not.

2.*4.1 parseToken() and parseTokenList()*

Two functions were designed in the token class that are essential for the parsing step of the interpreter. These functions are parseToken and parseTokenList (along with their literal variants mentioned later).

On the most basic level, parseToken takes a single element of the program input then sends it through a series of conditional checks to determine the type of token it should be. For example, if the user inputs an expression that contains a "+", the symbol is passed into parseToken then if it matches any of the defined operators in the Operator class (the addition operator "+") the function instantiates a operator token holding the addition operator then returns it.

On a more complex level, parseTokenList was designed to receive a list of strings (The split up user input) then parse each string to a token and detect nested lists of strings based on parenthesis. This is completed by using the parseToken function and recursive calls. The function works by searching for an opening parenthesis then parsing everything to a token until the next closing parenthesis. If another opening parenthesis is found while parsing then a recursive call to the parseTokenList function is made on the contents of the discovered parenthesis pair. The final result is a single token-list token which contains single tokens (atoms) or other lists of tokens (lists and function calls that require multiple arguments).

2.4.2 Implementing Quotation LISP Construct; parseLiteralToken() and parseLiteralTokenList()

Two functions were designed based on the previously discussed parseToken functions to handle cases where the user wants to use quotation to specify tokens as literals; these functions have the same implementations as the original two. However, each string sent to the functions are only parsed as string tokens with the literal boolean set to true. This prevents numbers and expressions from being evaluated. To detect when a user wants to quote an atom or list a conditional check is used in the original parseTokenList() to detect instances of (‘) and (‘( ). If detected the loop continues using the literal based parse functions recursively rather than the original.

2.5 **Evaluator.Java**

The Evaluator class structures an evaluator object which takes a token holding a LISP expression. The evaluator object then evaluates the expression using a series of conditional statements, Operator class logic, a stack, along with recursion then finally returns a token equal to the result. The evaluator class holds the implementation of the definition, constant literal, variable reference and function call LISP constructs (define, set! etc). Overall, the Evaluator class paired with the Operator class are complete the “Function call” LISP construct.

2.5.1 evaluate()

The evaluate function in the evaluator class takes a token and a hashmap then returns a token equal to the result of the initial token’s evaluation.

The function begins with a conditional that checks if the passed token is a literal or not. Since literals should not be evaluated, if the token is literal the function immediately returns the token. Otherwise the function continues.

Next, a stack is created which contains each of the tokens in the sent expression. The stack is structured where the first token to be popped off of it will always be an operator. If it is not an operator an exception is thrown. If there are no arguments once the operator is popped an exception will be thrown.

The next step uses recursion to ensure each of the tokens in the stack are evaluated and reduced to their simplest form before continuing the evaluation. For each token added to the stack, it is checked if the token is a token list. If the token is not a token list it remains on the stack in its current form. However, if the token is a token list, a new evaluator object is created then passed the list to run a recursive call to evaluate to retrieve the answer. The resulting token then replaces the discovered list in the stack.

Recursion is useful in this process because every nested LISP expression will be evaluated before an evaluation is attempted on the main operation found on the top level of the recursion.

This section will describe what takes place in each of the recursive calls . First, the operator initially popped off the stack is used as a parameter in a Java switch statement. The switch has cases for actions to perform based on the type of the operator (Types: arithmetic, boolean, lisp).

2.5.1.1 Arithmetic Operator Cases

If the operator is an “arithmetic” operator, a series of conditionals is used to determine its specific type. It is first checked if the operator is “sqrt”. If the operator is sqrt, conditionals are used to analyze the stack and determine if the appropriate arguments are provided for the sqrt function (In this case, a singular number token). If the stack is not appropriate, an exception is thrown. Otherwise, the number token is sent to the Operator class’ evalArithmetic function to perform the sqrt. The Token containing the sqrt result is then returned from the evaluator function.

Using the same process for evaluation sqrt, there are conditional cases for the “pow operator” as well as a general case for every other arithmetic operator such as “+” and “- ” which accept an arbitrary amount of numbers. For “pow” there is a check to ensure the stack consists of two number tokens before attempting to evaluate in the Operator class. For the general arithmetic operations the stack is evaluated to ensure that every token added to the stack is a number token; otherwise an exception is thrown.

2.5.1.2 Boolean Operator Cases including (and ,or, not)

If the operator is a “boolean” operator, there are conditional cases checked for if the operator is equal to “and”, “or” or “not”. If the operator is “and”, and there is a boolean token in the stack which is false the “and ” condition fails and a false boolean token is returned; otherwise a true boolean token is returned.

If the operator is an “or ” operator, the stack is checked for any instance of a true boolean token. If a true token is found a true token is returned from the evaluate function. Otherwise, the “or” condition fails and a false token is returned.

If the operator is “not”, a token which is opposite to the boolean token on the stack is returned; Hence if the stack has a true boolean token, a false token is returned and vice versa.

For the remaining boolean operators such as “<, >, !=” the conditional checks is every token on the stack is a number token viable for comparison. If the stack is not appropriate an exception is thrown. Otherwise, all of the stack’s number tokens are sent to the Operator class’ evalBool function to be evaluated for the given operator. Finally, the resulting boolean token is returned by the evaluate function.

2.5.1.2 Lisp Operator Cases including (define and set! constructs)

If the operator is a "lisp" operator, the first conditional checks if the operator is equal to "car" or "cdr". If the operator is "car" or "cdr", the stack is analyzed to ensure that the stack contains a token list token to perform the functions on. If the stack is not valid an exception is thrown. Otherwise, the token list is sent to the Operator class' evalLisp function where a token list token containing the result is returned from the evaluate function. The same process is used in the "cons" and "if" operator conditional cases. However, the stack is analyzed to ensure that the first token is a literal token and the second token is a token list token to add the first token to in "cons". For the "if" conditional the stack is analyzed to ensure that the first token is a boolean token, the second and third tokens in the stack are literal tokens (one of the literal tokens will be returned from the evalLisp call and from the evaluator function).

2.5.2 Implementing Define and Set! LISP Constructs

To implement the "define" and "set!" constructs, a Java Hashmap object was created in the interpreter class as a global variable to be passed into the evaluate function. The hashmap maps a string to a token object.

An example of how the hash map is used is shown in the "define" operator case. When this case is hit, the stack is analyzed to ensure that the first token is a string token and the second is a token. When the condition is passed a hashmap entry is created where the string is used as the variable name and the provided token is the token the variable name is mapped to.

To make use of variable definition and assignment in the previously discussed evaluate function operator cases, a conditional is placed at the beginning of each case to determine if any previously defined or assigned variables are being used in the current argument. If a variable is found in the hashmap the variable name token in the stack is replaced with its associated token value.

A checkHashMap function that takes a variable name token and the hashmap is used to return the specified variable token to the evaluate function.

3 CLASS CONCEPTS

A few class concepts I experienced when creating the interpreter include: local and global scoping, recursion, lexical analysis, syntax analysis and garbage collection.

Relating to the class concepts on the inner workings of compilers, this project explored some of the concepts and practices required for implementing a lexical and syntactical analyzer. The Parser, Token and Evaluator classes work together in this interpreter project to tokenize user input and error check its syntax. Additionally local and global scoping was very relevant to the relationships between classes and for function executions such as define. The define and set! functions required the use of a globally scoped hashmap to be a possibility. Finally a connection between LISP and this LISP interpreter implementation in Java is the use of automatic garbage collection. In the various Java constructs used throughout the project the garbage collection was automatically completed.

CONCLUSION/THOUGHTS

Overall I feel this project was a great experience to gain knowledge and experience with LISP. The project promotes research into the way LISP was designed in order to successfully implement the constructs. I also enjoyed working through ideas for structuring the project and Java classes in a way to allow for easy modification usability. I had fun taking on some of the challenges this final project posed and feel satisfied with the LISP concepts I have been able to implement

REFERENCES

I referenced a previous CSE213 assignment for an RPN calculator to get inspiration for some of the input parsing and tokenization.