Lisp Interpreter in JAVA

Dhrubo Saha
New Mexico Tech
Department of Computer Science
801 Leroy Place
Socorro, New Mexico, 87801
dhrubo.saha@student.nmt.edu

ABSTRACT

In this paper, I describe the simple implementation of Lisp interpreter in Java programming Language applying Stack technology. The goal of this project is to learn how the lisp works with static and dynamic scoping as well. So far I developed static scoping here.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Simplicity, Control Structures, Data Types & Structures, Syntax Design, Typing system, Interpreter, Exception Handling, Support for Abstraction (system & user levels)*

General Terms

Java, Stack, Lisp, Design

Keywords

Hash Map, Regular Expression

1. INTRODUCTION

Lisp is the second-oldest programming language that is still in widespread use (the oldest is FORTRAN). Lisp's simple and uniform syntax, interpreter, and ability to generate new data structures at runtime have encouraged the development of powerful programming environments that are best available for any language. I tried to develop the basic lisp syntax. The main goal of this project is to be familiar with some important language design features, being introduced with Lisp interpreter and also the JAVA language. Based on the project requirement I divided my project in three phases like 1.Parser 2.Lexical analyzer and 3.Interpreter. I am going to describe the phases in the following sections with the limitations I have in the project.

2. DESIGN

My design target was to implement an interpreter which will be user friendly like the regular Clisp works. Although I couldn't develop much more functionality like auto suggestions for the user flexibility but I am planning to develop that in my next phase. Right now, user can provide input one after another and based on the input output will show and also write to a output for the further necessity. After providing the lisp command I start to execute the processes one after another about what I provided a glimpse in the introduction.

2.1 Parser

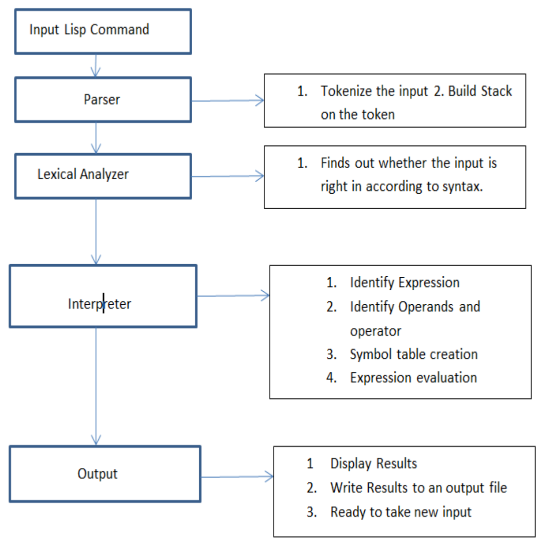
As we know parser is a compiler/interpreter component that breaks data into smaller elements for easy translation into another language. I tried to implement the parser here. First of all I tried to place space ( ) before and after of the parenthesis using the regular expression. Otherwise, it might be tough to tokenize perfectly. As an example, tokenize uses the empty space between each words to make every token. So as an example if it finds like: '(if' then it will merge these two as a single token which will be wrong to the next phase like lexical analyzer and that will show wrong information although having a syntactically correct expression. As we know the second task of a parser to build up a parse tree for which I used the Stack data structure.

2.2 Lexical Analyzer

In this phase I tried to check whether the provided input lisp command is syntactically correct or not. As in the parser phase I already tokenize the command and then push it to the stack, now during the evaluation I am going to pop out every token one by one. If I find any close parenthesis ( )) then my system is looking for another corresponding open parenthesis (( ). So if there is any kind of extra parenthesis open or close in the stack I am counting this as a syntax error and showing wrong output in the terminal. Then my system is going to be ready to take another command.

2.3 Interpreter

After checking whether the input command is syntactically correct, I am going to evaluate the command. Right now, I am taking care of couple of instructions like, mathematical expressions (add, subtract, multiplication and division), some build in functions (sin, sqrt, cos, tan), condition (if else), some built in key words like define, key and another most important thing like user defined function. To build up this user defined function we are using lambda as a keyword to denote the input text to be treated as a new function. There is a function which is checking every token to know their pattern and based on that, an stack is sending to the correspond function of the pattern. For example, if it finds 'if' keyword, then it is going to send the stack with 3 values what is given through the command, and based on the condition it is going to pop all the values from the stack and replacing the right values in the stack what is meaningful after executing the if command. Same thing is happening for all other commands. There is some kind of exception for the quote command. Because, if it calculate the values in the order of the stack, then whenever my function get the quote command, then it has nothing to print except the result but which is not meaningful for the quote command. For example : quote (+ 2 (+ 2 2)) , in this command, if its starts to execute the commands based on the stack order, then whenever it will get the quote command in that time, the mathematical values is going to calculated to 6 already. So it is going to print 6 only without printing the whole expression. So for this case I have check in the first place whether there is any kind of quote keyword exists in the command or not. If not then it will go to the default functionality of the program, but otherwise, I am going to print all the expressions without evaluating.


Input Lisp Command
Parser
1. Tokenize the input 2. Build Stack on the token
Lexical Analyzer
1. Finds out whether the input is right in according to syntax.
Interpreter
1. Identify Expression
2. Identify Operands and operator
3. Symbol table creation
4. Expression evaluation
Output
1 Display Results
2. Write Results to an output file
3. Ready to take new input

2.4 Output

The interpreter evaluates the expression and sends the result to the display and also writes the output to a file and then be prepared to take another input expression

3. IMPLEMENTATION

In this implementation phase, I have used couple of interesting features of JAVA which made the implementation quite easy. For the parsing purpose, I used the java tokenize function which treats everything after an empty space as a token and push it to the stack. Then the stack is passed to the interpreter. I also used the hashmap feature of JAVA which made me easy to keep several kinds of variables in memory and easy to use when it required.

3.1 Interpreter Implementation

The interpreter has some conditional statements. Every conditional statement looks for a particular keyword. If it matches then the corresponding action executed to the stack. For instance, (define r 10), this statement contains define keyword, for which it will go to the defvar function what I designed to push the 10 in the global stack 'tokenStack'. I also designed a Hashmap naming 'valueList' which is helping to mapping the value of 'r' in the hashmap.

Now again for instance (r) , in this command my system will check whether there is any kind of value for this in the hashmap and if it matches then returns the value of 'r'.

Same logic goes for all the other conditions, like (quote (+ 2 3)),

In this lisp command, according to the usual stack implementation it will start to compute (+ 2 3) which will not work for the quote command. So for this I had to search in the first place whether there exists 'quote' or not. If exists then it diverted from the usual stack implementation.

Another example can be:

(if (> 10 20) (+ 1 1) (+ 3 3)) in this expression, according to the stack, it will first evaluate (+ 3 3) and then push the value to the stack again. And then evaluate the command, (+ 1 1) and push the value as well. Finally it will also evaluate, (> 10 20) and push the true or false in the stack. Finally it will get the 'if' command, and based on the definition of 'if' condition, it will pop all the 3 elements (true/false, 2, 6) and if true then push 2 to the stack and if false then push 6 to the stack.

Using the stack it made so easy to implement the arithmetic expressions as well.

Like (+ 2 3), it will check the (+) and then adds 2 and 3 and push it to the stack again. During checking + and also adding 2 & 3 it will pop the elements from the stack so that it will not make any wrong sequence of the elements in the stack.

Nested expression is also an easier implementation for using Stack as the same process will be followed by interpreter and evaluated the expression and pushed to the stack as well.

3.2 User Defined Function Implementation

This was the most challenging issue in all over the project. Because, in this phase I am going to define a function through the command prompt and my interpreter will remember the function definition at runtime. And later, with any kind of that function call will evaluate the function definition what I stored in my program. For instance, (lambda ADD (x y) (+ x y))

Through this command, my interpreter will understand, user wants to define a function to use later. To implement that, as per the instruction we used the lambda function which will indicate to store a function name with the function definition as well. Like the 'quote' command we are facing also an issue here, as for the stack implementation, the usual way will start to evaluate, (+ x y) when there is not variable definition in the stack. So we had to divert from the normal implementation. If my interpreter find out the lambda keyword in the lisp command then it will go to the 'lambdaExpressionHandling' function to handle the corresponding functionalities like, first all we had to find out the function name from the lisp command, then through the regular expression we found out which will be the function parameter. In the given example, there are two parameter x & y. Then we also found out the function definition which is in this case: (+ x y)

To remember these things I used two more hash maps: 'lamdaParameter' and 'lamdafunctionDefinition'. 'lamdaParameter' keeps the all the parameter in a array with the mapping to the function name and 'lamdafunctionDefinition' keeps the function definition as a string with the mapping to the function name as well.

Now if the user calls this function with the parameter like : (ADD (8 3)) , then interpreter will try to find out whether there are any function in these two stacks or not. IF yes, then it will get the variable list from the 'lamdaParameter' and compare with the given parameter. If the number of the parameter matches then it move to the next phase which is to get the function definition from the hash map 'lamdafunctionDefinition' and then assigns the value to the corresponding variable and evaluates the expression like the previous usual stack implement function and push the final value to the push again to be printed.

4. FEATURES IMPLEMENTED

I have implemented all the features mentioned in the project file except the extra credit. The features have been implemented is listed below:

Variable Reference	Implemented
Constant Literal	Implemented
Quotation	Implemented

Conditional	Implemented
Definition	Implemented
Procedure Call	Implemented
Assignment	Implemented
Procedure	Implemented

The following predefined functions are also implemented:
(sqrt (x)) [Calculates the square root of x]
(sin (x)) [Calculates the sin of x]
(cos (x)) [Calculates the cosine of x]
(tan (x)) [Calculates the tan of x]

5. TEST CASES

a. Defining a new variable
(define r 10)
Output: r

b. View the value of a variable
(r)
Output: 10.0

c. View constant literal
(2)
Output: 2.0

d. Quotation
(quote (+ 1 2))
Output: (+ 1 2)
(quote (+ a b))
Output: (+ a b)

e. If
(if(> 10 20)(+ 1 1)(+ 3 3))
Output: 6.0

f. If with variables
(define a 3)
Output: a
(define b 2)
Output: b
(if (< a b) (+ 1 1) (+ 3 3))
Output: 6
(if (> a b) (+ 1 1) (+ 3 3))
Output: 2.0
(define x 2)
Output: x
(define y 2)
Output: y
(if (< a b) (+ x y) (- x y))
Output: 0.0
(if (> a b) (+ x y) (- x y))
Output: 4.0

g. Predefined function
(sqrt(4))
Output: 2.0
(sin(90))
Output: 0.8939966636005579
(cos(90))
Output: -0.4480736161291702
(tan(90))
Output: -1.995200412208242

h. Predefine function with variable
(define x 2)
Output: x
(sqrt(x))
Output: 1.4142135623730951
(sin(x))
Output: 0.9092974268256817
(cos(x))
Output: -0.4161468365471424
(tan(x))
Output: -2.185039863261519

i. Assignment
(define r 10)
Output: r
(set! r 5)
Output: 5.0
(r)
Output: 5.0
(set! r (* 2 2))
Output: 4.0
(set! r (* r r))
Output: 16.0
(r)
Output: 16.0
(set! r 2)
Output: 2.0

j. Arithmetic operation
(+ 2 3)
Output: 5.0
(- 3 2)
Output: 1.0
(* 2 3)
Output: 6.0
(/ 6 2)
Output: 3.0

k. Arithmetic operation with variables

```
(define a 5)
Output: a
(define b 10)
Output: b
(+ a b)
Output: 15.0
(- b a)
Output: 5.0
(* b a)
Output: 50.0
(/ b a)
Output: 2.0
```

l. Nested arithmetic operations:

```
(+ (+ 2 3) 5)
Output: 10.0
(- (- 2 3) 5)
Output: -6.0
(* (* 2 3) 5)
Output: 30.0
(define x 2)
Output: x
(define y 4)
Output: y
(define z 8)
Output: z
(+ (+ x y) z)
Output: 14.0
(* (* x y) z)
Output: 64.0
(- (- x y) z)
Output: -10.0
```

m. User defined function:

```
(lambda ADD (x y) (+ x y))
Output: ADD
(ADD (2 3))
Output: 5.0
(lambda SUB (x y) (- x y))
Output: SUB
(SUB (3 2))
Output: 1.0
(lambda MUL (x y) (* x y))
Output: MUL
(MUL (2 3))
Output: 6.0
(lambda DIV (x y) (/ x y))
Output: DIV
(MUL (6 2))
Output: 12.0

(lambda ADD (x y z) (+ (+ x y) z))
Output: ADD
(ADD(2 2 2))
Output: 6.0
(lambda MUL (x y z) (* (* x y) z))
Output: MUL
(MUL(2 2 2))
Output: 8.0
(lambda SUB (x y z) (- (- x y) z))
Output: SUB
(SUB(2 2 2))
Output: -2.0
```

6. CLASS TOPICS APPLIED

To develop this interpreter I needed several concepts which had covered in the class lectures. For instance, how the lisp command works, the prefix notation of the command, several kinds of functions were discussed in the class. Again, as I didn't implement dynamic scoping, to implement static scoping was quite easy in stack. Because, in the stack, whenever I am using the value then again pushes the updated value to the stack and the hashmap as well for which it helps to implement the static scoping quite easily.

7. LIMITATIONS

As I tried to implement a simple interpreter for the lisp, there are lot of functions which are not covered by my interpreter. Dynamic scoping is not developed here. Moreover, some exceptions are not handled perfectly so far, for which if there are several kind of deviation in the lispcommand, the program might face exceptions. But if lisp supported command is given in the interpreter then my interpreter will work good except some functionalities are not developed yet.

8. CONCLUSION

The purpose of this project was to understand the procedure to implement an interpreter of a programming language. In addition during the implementation of this interpreter I had to use some more exciting features of JAVA like Stack, Hash map. With using these structures I came to know that, this is so user friendly to work in JAVA. However, in case of understanding Lisp, I learned that, Lisp is very powerful procedural language. Moreover, the language bit complex comparing with the most prominent languages like C++, JAVA or Python.

9. REFERENCES

[1] http://stdioe.blogspot.com/2012/01/lets-implement-simple-lisp-interpreter.html