Building and Designing a LISP Interpreter in Java

Sterling Blevins
New Mexico Institute of Mining and Technology
Department of Computer Science
801 Leroy Place
Socorro, New Mexico, 87801
sterling.blevins@student.nmt.edu

ABSTRACT

This paper covers the process of development behind building a LISP interpreter with Java, and the concepts used to create it. Java is an object-oriented programming language with a great deal of flexibility. The entire program runs on the Java virtual machine level and is abstracted from the hardware. The overall design of the interpreter is made up of a *parser*, a *tokenizer*, *analyzer*, and a *processor*, which processes each statement. All expressions in LISP can be thought of as *numbers*, *keywords*, *operators*, and *variable* names. The interpreter is functional, with some minor error checking, and has provided an in-depth learning experience for designing and looking at the fundamentals that go into language design.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: *ALL*

General Terms

Documentation, Performance, Design, Reliability, Security, Standardization, Languages, Theory, Experimentation

Keywords

Languages, Development, Design, Java, LISP Power, Security, Robustness, Platform Independence, Readability, Maintenance, Efficiency

1. INTRODUCTION

The goal for this interpreter was to have a greater understanding behind how languages are designed and processed, in addition to understanding the underlying structure of an interpreter or compiler. In the case of this LISP interpreter, the language expressions are first parsed by a *parser*, and each part is then turned into tokens by a *tokenizer*. The separate tokens, are then processed and analyzed by an *analyzer*, then separated based on the token type, and then sent through a *processor* that will execute the entire expression. Each of these main components of the interpreter are supplemented by the *robustness* of error handling, the *speed* and *portability* of execution on the Java Virtual Machine (JVM) and the *power* of flexibility based on built in functions and the ability to change scoping mechanisms. Each supplement increases the functionality of the interpreter, but also has possible drawbacks. The LISP interpreter was largely a successful development project with a few minor road blocks along the way.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2017 NMT CSE324 - 5/3/2017

2. STRUCTURE & IMPLEMENTATION

The interpreter is loosely based on the structure of a compiler, in that it parses, analyzes, and executes each command. The object-oriented capabilities in Java made this easier to implement, because each object could be modified and altered for each statement and function in LISP.

2.1 Parsing the Input

The parser splits the user input into readable statements, divided by the parenthesis, and then further by spaces. The input is parsed by reading in the prompt input line by line. The LISP language contains a large amount of parenthesis. The LISP expression is read in and then is checked for parenthesis mistakes before tokenizing. The checker replaces accidental spaces before or after parenthesis and creates a standard input to be processed.

```
input = input.replaceAll("(\\s+([(]))", "(");
input = input.replaceAll("(\\s+([)]))", ")");
```

Figure 1. Parenthesis Checking

The expression entered by the programmer is then split using the regular expression [()], so as to split the input into an array of statements, divided by the parenthesis. For each individual statement, the array is then handed off to the Tokenizer.

2.2 Tokenizer

The tokenizer has the ability to translate input into four possible types: a number, a variable name, a keyword, or an operator.

2.2.1 Keywords

The keywords library consists of an enumeration with a corresponding string. All keywords are the corresponding names to the built in functions

Table 1. Keyword Library

Keyword	*Description*
`quote`	Returns the expression literally
`if`	Conditional statement
`define`	Define a variable
`set!`	Set is a redefinition of a variable if it exists
`defun`	Define the contents of a function
`exp`	Takes two values, computes the exponential value of the first value with the second being the exponent
`cos`	The radians equivalent of cosine of a value
`sin`	The radians equivalent of the sine of a value
`tan`	The tangential equivalent of the tangent of a value
`sqrt`	The square root of a value

When a token is one of the last five keywords, the enumeration class can calculate the value using Java's Math library.

2.2.2 *Operators*

The operators library contains all operators made available within the interpreter. The library also contains the conditional operators for the 'if statements' above.

Table 2. Operator Library

Operator	*Meaning*
(	Left parenthesis
)	Right Parenthesis
^	Another way to use exponents
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction
>	Greater than
<	Less than
<=	Less than OR equal to
>=	Greater than OR equal to
==	Equal to

The operator class enumeration can evaluate based on which operator is used and how many operands it uses

2.2.3 *Variables or Numbers*

The variables are any characters or strings that are not matched by a number, keyword, or operator. A number is a value defined as a double.

2.2.4 *Tokenizer*

The actual Tokenizer mainly consists of a large 'if-else statement' comparing the string to every possible operator, keyword, or if nothing matches, a number or can be declared a variable name. Each token is then thrown into a list and processed.

2.2 Analyzer

The analyzer serves as a directory for the type of execution, depending on which type of token it is. The analyzer takes in the array list of token lists. The analyzer calls in the list, and the index position. The program starts off at index position 0. The first string is checked to see if it is a number, a variable, an operator, or a keyword call. The processing type depends on the token. The LISP Interpreter required the implementation of 8 expression types that the program was supposed to be able to interpret.

2.3.1 *Value / Variable Reference / Constant Literal*

The value of an equation can be computed using prefix notation. Check the *Table 2* for possible operators. A number that is entered will simply return a value, and a variable (assuming it has been defined) will return the value of the variable.

Table 3. Value / Variable Reference / Constant Literals

Token	Syntax	Result
operator	`(+ 3 4)`	7
number	`7`	7
variable	`r`	10

2.3.2 *Variable definition*

The variable definition defines a variable 'r', and assigns it to a value '10'.

Table 4. Variable Definition Syntax

Keyword	Syntax	Result
define	`(define r 10)`	r > 10

2.3.3 *Quotation*

The quotation restates the literal term and does not evaluate the (+ 1 2) equation.

Table 5. Quotation Syntax

Keyword	Syntax	Result
quote	`(quote (+ 1 2))`	(+ 1 2)

2.3.4 *Conditional*

The 'if statement' evaluates if the test, (< 10 20), and if it proves true, the first statement is evaluated. If false, the second statement is evaluated.

Table 6. Conditional Syntax

Keyword	Syntax	Result
if	`(if (< 10 20)(+ 1 2)(-3 4))`	3

2.3.5 *Procedural Call*

There are built in procedure calls that perform special mathematical operations.

Table 7. Procedure Call Syntax

Keyword	Syntax	Result
cos	`(cos(90))`	
sin	`(sin(90))`	
tan	`(tan(90))`	
sqrt	`(sqrt(4))`	2
exp	`(exp(2, 4))`	16

2.3.6 *Assignment of a Variable*

The set function will assign a variable that has already been declared to a given expression.

Table 8. Set! Syntax

Keyword	Syntax	Result
set!	`(set! r (* r r))`	r > 10 > 100

2.3.7 *Function Definition ('lambda')*

The function definition will assign a set of parameters and expressions to execute to a function name. Once called, the function will be executed based on the provided parameters.

Table 9. Function definition Syntax

Keyword	Syntax	Result
defun	`(defun ADD (x y) (+ x y))` `(ADD (3 4))`	7

2.3 Processor

The processor methods are split up but interconnected. Each method takes in the entire array of token lists, the index, and the variable array map.

2.3.1 *Operators / Equations*

The operators are thrown through the `evaluation` method. If the first index value is an operator, the evaluation method is called. Within the evaluation method, the variables can be translated and swapped out with the corresponding number.

2.3.2 Variable Definition / Variables request / Set!

The variable definition takes the variable term, and the variable value and uses a Hash Table to store it. There are two hash tables. One for the variable definitions that are globally defined, and one for the variable definitions defined within a function.

The variables are defined within `variableDefinition` and if an equation ever requests a variable, they are converted using `variableExchange`, which just uses the hash map to replace the token.

2.3.3 Quotation

The quotations take everything after the keyword 'quote' and print accordingly. Beyond printing equations, it has quirks. Quotations are processed by the `quotation` method.

2.3.4 Conditional

The conditional statements are broken up into three parts. The test section will evaluate to either True (1) or False (0). Conditionals are only evaluated using the conditional operators (see chart). If the conditional proves true, only the first statement is evaluated. If the conditional is false, the last statement is evaluated. The conditional is index 1, the true case is index 2, and the false case is index position 3.

2.3.5 Procedural Calls

The procedural calls are built in math functions that act as short-cuts. The built in functions are translated by the method `math`. The procedural calls can compute equations within the bounds of their parenthesis too. The functions are processes are described in table…

2.3.6 Function Definitions / Function Calls

The function definition is created with a class if the keyword is `defun` with the method `defineFunction`. The class contains the name of the function, the possible parameters, and the expressions within the function. Each function class is placed into an array list of function classes. When a function is called, the function class list is checked if it exists. If it does, the class is processed, and executed with `executeFunction`.

2.3.7 Static / Dynamic

The static and dynamic scoping implemented for the extra credit allows the user to switch the Hash Mapping from two separate Access records in the static mechanism to using the same Access records for all variable definitions. This is all decided with `mapFlag` or `type`. See conclusion.

2.3.8 Command Result Record

This was implemented through a writer class call. For every call to the command line print out, a write to the file was processed. It is cleared or created every time the interpreter is run.

3. CONCLUSION

The overall project was difficult at first, but easy enough to understand after some extensive work. The project allowed us to grasp a greater understanding of how an interpreter works and functions, in addition to covering many topics discussed in class.

3.1 Connections to class topics

The included list of operators, and keywords allow for a mild amount of power. There are 10 possible keywords that can all be used together in one way or another. There are 13 possible operators, which also contribute to the power. Each library handles a subset of the execution which makes the interpreter somewhat efficient. Limited to 5 Java classes, the assignment is low cost, easily read and maintained. It is also highly portable on the JVM. Although the assignment had to be modified structurally to use a linear tree instead of a stack, it functions on at least two globally available version of Java.

There is a limited problem with security and robustness. The interpreter has a very limited error checking mechanism. The two main errors I ran into when trouble shooting the interpreter were null-pointer-exceptions and out-of-bounds-array-exceptions. Both will be ignored and the interpreter will keep running when a user inputs a malformed statement, but the user will not be warned specifically of what happened or how to fix it. Variables that are not defined are error checked and the user is warned.

3.2 Problems and Solutions

An original issue I ran into, as I mentioned before was portability. I attempted to use stacks in order to process each execution, but due to an issue with a revision in Java 1.8 (the latest JDK) over Java 1.7 (the version running on TCC computers) I had an issue with the compatibility of stack manipulations between versions. I changed to a list / flat tree manipulation and it functioned well between both versions.

I also ran into some issue with implementing the extra credit. I attempted to make it possible to switch between a static and dynamic scoping mechanism. This can be done by sharing the Access records or hash maps between the functions and global declarations. This is done with a `mapFlag`, with 0 or 1 being the global definitions, and 2 being shared. Unfortunately there were some issues that could not be overcome in the provided time frame.

4. REFERENCES

[1] "Enum Types." Enum Types (The Java™ Tutorials > Learning the Java Language > Classes and Objects). Accessed April 17, 2017. https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

[2] Java and Makefiles. Accessed May 01, 2017. https://www.cs.swarthmore.edu/~newhall/unixhelp/javamakefiles.html.

[3] Maes, Jessie. *Generic Classes and Data Structures*. PDF. Socorro, NM: NMT CSE 213, March 22, 2017. Homework 6 for CSE213

[4] Tutorialspoint.com. "LISP Tutorial." Www.tutorialspoint.com. Accessed May 04, 2017. http://www.tutorialspoint.com/lisp/.