Project 2: Implementing a Lisp Interpreter in C++

Julian Garcia
New Mexico Institute of Mining and Technology
Department of Computer Science
801 Leroy Place
Socorro, New Mexico, 87801
julian.garcia@student.nmt.edu

ABSTRACT
This report details my experience in writing a LISP interpreter in C++. In writing a Lisp interpreter, I was given a set of Lisp constructs to implement and the report details my progress with each construct. Ultimately I could not complete the project as intended, the interpreter submitted can only handle math functions readily.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: *ALL*

General Terms
Design

Keywords
HLL

1. INTRODUCTION
LISP is an older programming language with many unique features. LISP consists mainly of functional application commands. Besides function application, there are forms of assignment commands and conditional commands written in functional form. In general, recursion is used in place of iteration. An expression like "1.5 + 2" is a LISP statement which means to type-out the result of applying + to the numerical arguments 1.5 and 2. In LISP, prefix notation is used, e.g. +(1.5, 2). Moreover, instead of writing f(x, y) to indicate the function f taking the arguments x and y, we write (f x y) in LISP, so (+ 1.5 2) is the LISP form for "1.5 + 2".

LISP functions can also operate on lists of objects; which is where the name comes from, "LISP" is derived from the phrase "List Processing". LISP is implemented with an interpreter called the LISP interpreter. The LISP interpreter reads LISP expressions which are entered as input and evaluates them and types out the results. In this project, I attempt to implement a Lisp interpreter using the programming language C++.

My Lisp interpreter is very incomplete, and can only handle math operations. This is due to how I structured my work. In planning how I'd approach this project (after abandoning code after the second progress report), I decided to begin with what I figured would be the simplest to implement, math constructs. In doing so I had two objectives, the first and most important would be to create a template for creating functions that I could expand upon, the second would be creating a well rounded tokenizer/parser that could handle math functions within math functions. In my code you'll see that I created an array of functions, which contain each function usable in the emulated lisp environment. My tokenizer (GetNextToken) finds each significant lisp token and my parser uses those tokens to determine when a function needs to be run and what needs to be returned. My main function prints out the result of my parser function which returns a double. The biggest issue in my code comes in how I implemented my parser. Specifically, the fact that my parser returns a double inhibited my progress greatly. To fix this issue, I was planning on creating a hash function, to convert strings into unique doubles, so that my parse function could technically return strings as well. This is where I was stuck in my code however, as I only came to this conclusion after trying many different methods of forcing my parse function to return multiple types, which was a horrible waste of time. Since my parse function could not return strings, any non-math related operation got put on hold really. Detailed in the rest of this paper is the status of each lisp construct assigned in this project.

2 Variable Reference
2.1.1 Functionality
Variable referencing simply allows for symbols to be interpreted as values, for example we can have a var to have a value of 10. In Lisp, variables can be statically scoped, and dynamically scoped, depending on the syntax.

2.1.1 Implementation
To implement this in C++, my first idea was to use a two dimensional array to store each variable and each variable's value. In our implementation of a Lisp Interpreter, only static scoping is allowed thankfully. The array is defined globally and stores every variable "defined". Unfortunately, due to how I structured my work, this was the last structure that I was working on, and every variable defined always evaluates to 0, with the method for definition being along the lines of "c 14" as I had not yet implemented the function method for this.

3. Constant Literal
3.1.1 Functionality
The functionality of a constant literal is to ensure that numbers, booleans and other unchangeable values evaluate to themselves. Essentially to implement this we need to ensure that 5 will always equal 5. True will always equal True, Null will always equal Null and so on...

3.1.2 Implementation
To implement this, we essentially need to place checks in our code. Implementing this in my c++ code comes in when I'm parsing the expression for tokens. If the given expression only gives a constant literal without any parenthesis, the value of that literal will be returned.

4. Quotation

4.1.1 Functionality

Quotation in Lisp, using the single quote symbol as syntax (‘), makes sure that whatever argument associated with the quote isn’t evaluated, but is simply returned as itself. If we were to pass ‘(+ 1 2) we’d simply return (+1 2).

4.1.2 Implementation

To implement this we’d again have to run a check every time an expression is passed. To implement this in my C++ code, I was planning on adding a new token type to determine if the token was being quoted. However, before I would implement this, I needed to get more fundamental functionality established, such as simply parsing strings.

5. Conditional

5.1.1 Functionality

Simply evaluate one statement if a test statement is true, otherwise evaluate another statement. Syntax is as follows: “(if test conseq alt)”, where we see if “test” is true, then go to “conseq” if it is true and “alt” if otherwise.

5.1.2 Implementation

To implement this in C++, I first have to parse the code for an “if” statement, then I have to make sure that what follows the “if” statement follows the format given above, and return a syntax error if it doesn’t. In the middle of the semester I had this somewhat functional, however I abandoned that code to just establish a base in which I could expand upon, and so there are no conditionals in the code I submitted.

6. Variable Definition

6.1.1 Functionality

Variable definition allows for new variables to be created and given a value. In Lisp the syntax for this would be “(define var exp)” where “var” is the variable name and “exp” is the value to be assigned to.

6.1.2 Implementation

This is simple enough to implement in my C++ code, as for every new variable defined I simply have to add to my 2d array. To identify if a variable is defined I simply have to parse the text given by the user to see if “define” is used, I once again have to check for any syntax errors then finally add to the dynamic array of variables if the syntax is correct. Right now my code parses the 2d array when a simple character is given, and if it is found returns the value associated with the variable. This is incomplete, as every variable defined returns 0. Also, the amount of variables allowed in a session would be limited to a certain amount, currently only 20 variables would be allowed, as the 2d array used to store variables in my code is static.

7. Function Call

7.1.1 Functionality

Handling function calls is fundamental to establishing this Lisp interpreter. A function call simply passes a set number of arguments to a function to be evaluated and returned.

7.1.2 Implementation

To implement this in C++, I’ll parse the code to see if the function is “define” or “quote”, if it is not, I’ll treat the function call as its own function and search for a function which is associated with it. To do this, I need to have a list of function names to read from, if a function name is found, I can then use that name to run a defined function (otherwise the code will of course return an error). Right now, there is absolutely no functionality for this.

8. Assignment

8.1.1 Functionality

This allows for variables to be assigned new values. The syntax for this in Lisp is (set! var exp), where “exp” is the value to be assigned to the pre-defined variable “var”, we use the notation “set!” to indicate an assignment.

8.1.2 Implementation

I haven’t implemented this at all in my code. The idea would be to again search through that 2d array, and reassign the string value associated with that variable.

9. Function Definition

9.1.1 Functionality

Defining a function simply defines a block of code to be called upon and run using given arguments. The syntax for this in Lisp is (defun foo (var1 var2...) exp), where “defun” is used to indicate that a function is being defined, “foo” is the name of the function, “var1 var2” are the arguments of the function and “exp” would be the body of the function.

9.1.2 Implementation

Another construct I hadn’t gotten to. The general idea for this would be to add to the array of functions defined in my code, with a set limit on the amount of functions which could be defined at once.

10. Arithmetic Operators

10.1.1 Functionality

The arithmetic operators we’ll use in this interpreter are +, -, *, / and we’ll be allowing them on the integer type. The functionality of each are fairly straightforward, as “+” indicates addition, “-” indicates subtraction, “*” indicates multiplication, and “/” indicates division.

3.1.1 Implementation

This is the one construct I have completed in my code. My Idea again was to start with simple arithmetic, and expand from there. In making this construct work, I believe I harmed my progress as much as I helped it. I created a parse function which could completely evaluate math functions within math functions to make sure that I could apply this foundation to other constructs. What harmed my progress is that my code was only meant to handle doubles, and so every construct yet to be completed would have to work around this. So to evaluate strings I was beginning to make a hash function to turn each string into a unique double which would later be turned back into a string when needed.

11. Other Functions & Expressions

Since these Lisp constructs feel like they aren’t essential to establishing the fundamental usage of the Interpreter, they will be the last constructs I implement. I currently haven’t approached implementing any of these constructs at all, though implementing comparators and the construct “Sqrt, pow” seem like they’ll be simple to implement.

11.1.1 “Car” “Cdr”, “Cons”

Currently, the functions supported in my code are math based, though I feel as though implementing this would’ve been simple after I managed to make my parse function handle strings through a hash function. If I’d been able to do that, I would’ve passed any lists found as a hashed string, then I’d be able to run these list operations using that info. Since I’d simply add these functions to the array of functions currently in my code and implementation might've been as simple as creating another tokenizer.

11.1.2 Sqrt, exp

I managed to complete both of these entirely, which was a fairly easy process as my code was meant to expand upon math functions.

11.1.4 Comparators (<,>,==,!=, and, or, not)

The comparators: <, >, ==, != all somewhat work in my code right now. If the user enters (< 1 2) the output will be "1" to indicate true. This same format applies to the other operators, however to use '==' the user would have to input (= 1 2) instead of the full '==' input, this applies to != as well as the user would have to input (! 1 2). This is due to how I implemented the comparators, I treated them as simple operands, which are all denoted by a single character in my code.

12. SUMMARY

This was a very difficult assignment for me, especially since I was almost completely unfamiliar with actually coding C++ before attempting this project. However, I feel as though I learned a lot in attempting to complete this project. My biggest error was in focusing on math functions at the core of my work. Though I managed to setup a function template that would make implementing any math function easy, the issue arose in implementing anything not math related.

13. REFERENCES

[1] Bruce J. Maclennan, Programming language comparisons, general knowledge, system and language characteristics.

Principles of Programming Languages, Design, Evaluation, and Implementation 3rd edition. Oxford University Press. 1999.

Lisp Construct	Status
Variable Reference	Partially Done
Constant literal	Done
Quotation	not
Conditional	not
Variable Definition	Partially Done
Function call	Partially Done
Assignment	Partially Done
Function Definition	Partially Done
The arithmetic +, -, *, / operators on integer type.	Done
"car" and "cdr"	Not
The built-in function: "cons"	Not

sqrt, pow	Done
>, <, ==,!=, and, or, not	Not