Project 2 Progress Report: 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 so far 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 most of the work accomplished has been in planning out how each construct will be made functional in C++.

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++.

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 dynamic array to store each variable and each variable's value. In our implementation of a Lisp Interpreter, only static scoping is allowed thankfully. The dynamic array will be contained in a data structure which will be created to contain variables within their respective functions.

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 on our variable referencing. Implementing this in C++ essentially means running a true/false check on if the user is trying to use a constant literal (such as the boolean value "True") as a variable name. I accomplished this by essentially running type checks on each variable made by the user in their Lisp code.

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 have to run a check every time an expression is passed. To implement this in C++, I'm currently taking every expression as a string and I'm evaluating the first character in the string to check if it is a single quote. If it is a single quote, I'll parse the rest of the text to determine what the expression is then I'll return that expression literally.

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. If the statement does follow the correct syntax, my code evaluates the test statement to see if it is true or not. Then either conseq or alt is parsed, depending on the result of the test.

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 dynamic 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 I'm only looking at integers as created variables and haven't really begun testing strings.

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).

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 begun attempting to implement this yet in my code. My current plan to implement this involves searching the dynamic arrays for the variable name, then reassigning the value using that name.

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

I'm not sure if my current process for implementing this construct will remain in the final code. Put vaguely, my implementation searches for the "defun" indication, returns an error if the following syntax doesn't follow the format, then adds on to a dynamic structure with each variable defined within the function stored in the struct for that function.

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

Currently just parsing the text for each operand and simply performing the operation if they're found with correct syntax. Though I may change this so that each is treated as a function with the arguments to each being treated as actual function arguments within how my code treats functions.

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" and "Cdr"

Haven't begun attempting to implement this yet.

11.1.2 "Cons"

Haven't begun attempting to implement this yet.

11.1.3 Sqrt, exp

Haven't begun attempting to implement this yet.

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

Haven't begun attempting to implement this yet.

12. SUMMARY

So far, my Lisp interpreter is far from completion in terms of code. Most of my time working on getting a complete plan in mind in terms of how to meet every requirement of a Lisp interpreter to ensure that I don't have to waste time completely redesigning code due to a misconception in terms of how to make each Lisp construct function using C++.

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	Partially Done
Conditional	Done
Variable Definition	Partially Done
Function call	Not
Assignment	Not
Function Definition	Not
The arithmetic +, -, *, / operators on integer type.	Partially Done
“car” and “cdr”	Not
The built-in function: “cons”	Not
sqrt, pow	Not
>, <, ==,!=, and, or, not	Not