Lisp Interpreter

Project Report

Christopher Montoya
New Mexico Institute of Mining and Technology
Socorro, United States
christopher.montoya@student.nmt.edu

1 INTRODUCTION

The programming language I chose to create my lisp interpreter in was C++. I chose C++ because it is the language I am most comfortable with, although the lack of dynamic typing and manual memory management added a significant amount of difficulty to the project.

In my interpreter, I tokenize the user input into objects of a custom class called "Atom", taken from the actual Lisp definition of an Atom we covered in class, and then parse those tokens in a stream to evaluate the Lisp command.

2 IMPLEMENTATION TABLE

Lisp Construct	Status
Variable Reference	Done
Constant Literal	Done
Quotation	Done
Conditional	Done
Variable Definition	Done
Function Call	Done
Assignment	Done
Function Definition	Done
The arithmetic +, -, *, / operators on integer type	Done
“car” and “cdr”	Done
The built-in function: “cons”	Done
sqrt, exp	Done
>, <, ==, <=, >=, !=	Done

3 VARIABLE REFERENCE

3.1 Implementation

I knew going into the project that I wanted to implement variables in such a way that would mimic traditional static scoping. To do this I decided to create a class called "Environment" which would store each variable and its definition. Then I would store those Environments in another class called EnvironmentContainer to keep track of how the Environments are organized for proper variable referencing. Then I would simply search through the Environments for the proper variable reference.

The following screenshot shows an example of the code that deals with variable referencing.

```
Atom Parser::search_for_symbol(string symbol_name) {
    EnvironmentContainer* container = EnvironmentContainer::getInstance();
    Atom result = container->lookup(symbol_name);
    return container->lookup(symbol_name);
}
```

3.2 Class Topics

I got the idea of static scoping from the material we covered in class. I knew that static scoping would be more user-friendly and easier to understand for programming purposes so I decided to try and implemented that. I also knew that Lisp traditionally uses static scoping from the Lisp lecture we covered in class as well.

3.3 Additional Implementation Details & Screenshots

```
>(define x 10)
x
>(+ x x)
20
>(set! x 50)
x
>(+ x x)
100
```

```
>(x)
25
```

Variable referencing primarily uses the Environment and EnvironmentContainer classes to store and find the proper values for variables. The variables themselves are stored as instances of the Atom class with their corresponding definition as a key,value pair in an unordered_map. To use the unordered_map class, I used the standard C++ <unordered_map> library.

4 CONSTANT LITERAL

4.1 Implementation

To have numbers and booleans evaluate to themselves, all I had to do was have a check in my evaluation function that would return Atoms that were either numbers or booleans.

4.2 Class Topics

I knew that numbers and booleans, ie. literals, should evaluate to themselves from the Lisp lecture in class.

4.3 Additional Implementation Details & Screenshots

```
if (atom.is_number) return atom;
```

```
} else if (atom.token_val == TRUE || atom.token_val == NIL) {
    return atom;
```

This occurs in the evaluate() method in the Parser class, which parses the Atom tokens to evaluate the Lisp command.

5 QUOTATION

5.1 Implementation

To have quotations not be evaluated and instead just be taken as literals, I had to have a section in the Lexer class which will treat the section after a quote as a literal, and then store that literal as a literal Atom.

```
} else if (c == QUOTE) {
    int open_paren = 0, close_paren = 0;
    stringstream ss;
    string expression = "";
    i++;
    do
    {
        if (input.at(i) == LPAREN) open_paren++;
        else if (input.at(i) == RPAREN) close_paren++;
        expression = expression + input.at(i);
        i++;
    } while (open_paren != close_paren);
    stream.push_back(Atom(LITERAL, expression));
```

5.2 Class Topics

I understood the idea of literals and their purpose from the Lisp lecture in class.

5.3 Additional Implementation Details & Screenshots

```
>('(+ 1 2))
(+ 1 2)
```

```
} else if (atom.is_literal) {
    return atom;
```

I set the evaluation method to return the Atom automatically if it is a literal, since literals shouldn't be evaluated as instructions.

6 CONDITIONAL

6.1 Implementation

I put the conditional evaluation logic with the rest of my boolean evaluation code. It first evaluates the condition, then if it is true, it evaluates the first statement. If it isn't true, it evaluates the second statement.

```
case IF: {
    Atom evaluation = evaluate(stream);
    if (stream->peek().token_val == RPAREN) {
        atom = stream->eat();
    }
    if (evaluation.token_val == TRUE) {
        return evaluate(stream);
    } else {
        stack<char> parenthesis;
        atom = stream->eat();
        parenthesis.push(atom.token_val);
        while (!parenthesis.empty()) {
            atom = stream->eat();
            if (atom.token_val == LPAREN) {
                parenthesis.push(atom.token_val);
            } else if (atom.token_val == RPAREN){
                parenthesis.pop();
            }
        }
        return evaluate(stream);
    }
    break;
}
```

6.2 Class Topics

I saw examples of how the if statements should work from the in-class Lisp lecture.

6.3 Additional Implementation Details & Screenshots

```
>(if (< 3 2) (+ 10 10) (- 0 10))
-10
>(if (< 2 3) (+ 10 10) (- 0 10))
20
```

In the first statement, 3 < 2 is not true, so the else section activates and evaluates - 0 10, which is -10

In the second statement, 2 < 3 is true, so the then section activates and evalutes + 10 10 as 20.

7 VARIABLE DEFINITION

7.1 Implementation

To define a variable, first it has to be defined with the 'define' keyword, a name, and either an expression or a value. Then the variable

definition is stored in an Environment object which is then stored in the EnvironmentContainer.

```
Atom Parser::define(InputStream *stream) {
    string variable_name = stream->eat().value;
    Atom evaluation = evaluate(stream);
    Environment env;
    env.add_symbol(variable_name, evaluation);
    EnvironmentContainer* container = EnvironmentContainer::getInstance();
    container->push_environment(env);
    return Atom(LITERAL, variable_name);
}
```

7.2 Class Topics

The implementation of static scoping and environments was taken from the lecture about scoping in class, as well as the fact that traditional Lisp uses static scoping. I tried to have a system that would emulate the contour diagrams we used in class to display static scoping.

7.3 Additional Implementation Details & Screenshots

```
>(define x 10)
x
>(+ x x)
20
>(set! x 50)
x
>(+ x x)
100
```

8 FUNCTION CALL

8.1 Implementation

Function calls are done through searching through the EnvironmentContainer object for the function definition. Then the actual parameters are evaluated and then replace the formal parameters in the function definition. Then the resulting definition is tokenized and evaluated.

```
Atom Parser::eval_function(Atom function, InputStream *stream) {
    vector<Atom> actual_parameters;
    while (!stream->is_empty()) {
        stream->print_tokens();
        Atom evaluation = evaluate(stream);
        cout << evaluation.value << endl;
        actual_parameters.push_back(evaluation);
    }
    vector<string> formal_parameters = function.formal_parameters;
    string operation = function.operation;
    int index = 0;
    for (string formal_parameter : formal_parameters) {
        operation = regex_replace(operation, regex(formal_parameter), actual_parameters.at(index).value);
        index++;
    }
    Lexer lexer;
    InputStream function_stream = lexer.tokenize_input(operation);
    return evaluate(&function_stream);
}
```

8.2 Class Topics

I got the idea of actual and formal parameters from the material covered in class. Separating the parameters into the formal and actual sections made it easy to replace the formal parameters with the actual parameters and then evaluate the expression of the function definition.

8.3 Additional Implementation Details & Screenshots

Once a function is defined, it can receive parameters and will return the calculated result.

```
>(ADD (10 15))
25
```

9 ASSIGNMENT

9.1 Implementation

Variables can be assigned new values after they have already been defined. This is done through the set method in the Parser class, which will update the value of the variable in the EnvironmentContainer.

```
Atom Parser::set(InputStream *stream) {
    string name = stream->eat().value;
    Atom atom = search_for_symbol(name);
    if (atom.token_val == UNDEFINED) {
        return atom;
    }
    Atom evaluation = evaluate(stream);
    EnvironmentContainer* container = EnvironmentContainer::getInstance();
    Environment env;
    env.add_symbol(name, evaluation);
    container->push_environment(env);
    return evaluation;
}
```

9.2 Class Topics

The implementation of set! was inspired by the Lisp lecture in class.

9.3 Additional Implementation Details & Screenshots

```
>(set! x (+ 25 25))
x
>(+ x x)
100
```

10 FUNCTION DEFINITION

10.1 Implementation

Function are defined by using the 'defun' keyword. The user must specify the function name, the formal parameters, and the operation that will be performed by the function. This is implemented by the defun method in the Parser class.

```
Atom Parser::defun(InputStream *stream) {
    string function_name = stream->eat().value;
    vector<string> formal_parameters;
    string operation = "";
    Atom atom = stream->eat();
    while (atom.token_val != RPAREN) {
        if (!atom.is_token) formal_parameters.push_back(atom.value);
        atom = stream->eat();
    }
    atom = stream->eat();
    do
    {
        if (atom.is_token) {
            operation = operation + static_cast<char>(atom.token_val) + " ";
        } else {
            operation = operation + atom.value + " ";
        }
        atom = stream->eat();
    } while (!stream->is_empty());
    Atom function_def = Atom(FUNCTION, formal_parameters, operation, function_name);
    Environment env;
    env.add_symbol(function_name, function_def);
    EnvironmentContainer* container = EnvironmentContainer::getInstance();
    container->push_environment(env);
    return function_def;
}
```

Then, the function and its definition is stored in an Environment so the definition can be retrieved later.

10.2 Class Topics

Once again, the idea of formal parameters and statically scoped functions was taken from the Lisp lecture, as well as a few other class lectures.

10.3 Additional Implementation Details & Screenshots

```
>(defun ADD (a b) (+ a b))
ADD
>(ADD (10 50))
60
```

11 ARITHMETIC

11.1 Implementation

Arithmetic is evaluated in the eval_arithmetic method in the Parser class.

```
Atom Parser::eval_arithmetic(Atom atom, InputStream *stream) {
    switch(atom.token_val) {
        case PLUS: {
            int operand1 = stoi(evaluate(stream).value);
            int operand2 = stoi(evaluate(stream).value);
            atom = stream->eat();
            if (atom.token_val == RPAREN) {
                return Atom(NUMBER, to_string(operand1 + operand2));
            }
            break;
        }
        case MINUS: {
            int operand1 = stoi(evaluate(stream).value);
            int operand2 = stoi(evaluate(stream).value);
            atom = stream->eat();
            if (atom.token_val == RPAREN) {
                return Atom(NUMBER, to_string(operand1 - operand2));
            }
            break;
        }
        case MULTIPLY: {
            int operand1 = stoi(evaluate(stream).value);
            int operand2 = stoi(evaluate(stream).value);
            atom = stream->eat();
            if (atom.token_val == RPAREN) {
                return Atom(NUMBER, to_string(operand1 * operand2));
            }
            break;
        }
        case DIVIDE: {
            int operand1 = stoi(evaluate(stream).value);
            int operand2 = stoi(evaluate(stream).value);
            atom = stream->eat();
            if (operand2 == 0) {
                puts("Error! Attempted division by 0");
                return Atom(ERROR, "ERR");
            }
```

This method recursively calls the evaluate method and to evaluate the operands and then uses C++'s in-built arithmetic to calculate the result. The result is then converted to an Atom and then returned. It has error checking for division by 0.

11.2 Class Topics

From the lectures in class, I knew that Lisp uses pre-fix notation, this was actually easier to implement in this case as pre-fix notation lets me know what operation needs to be performed and I can leave the evaluation of the operands to the recursive calls.

11.3 Additional Implementation Details & Screenshots

I use several C++ functions like stoi() and to_string() in order to easily convert the numbers from their string form to integers.

```
>(+ (+ 15 10) 40)
65
>(- 10 20)
-10
>(* (+ 5 5) 2)
20
>(/ 10 0)
Error! Attempted division by 0
ERR
```

I also included a check to make sure that the inputs are within the bounds of the int data type.

```
>(+ 3000000000000 5)
INTEGER OVERFLOW ERROR
```

12 "CAR" AND "CDR"

12.1 Implementation

car and cdr are implemented in methods in the Parser class. car returns the first element of a list, while cdr returns everything but the first element. The implementation of each is very similar in structure, but the logic is quite different.

```
Atom Parser::car(InputStream *stream) {
    int open_paren = 0, close_paren = 0, count = 0;
    Atom atom = evaluate(stream);
    string expression = atom.value;
    expression = expression.substr(1, expression.size() - 2);
    string first_element = "";
    do
    {
        first_element = first_element + expression.at(count);
        if (expression.at(count) == LPAREN) open_paren++;
        else if (expression.at(count) == RPAREN) close_paren++;
        count++;
    } while (open_paren != close_paren);
    return Atom(LITERAL, first_element);
}

Atom Parser::cdr(InputStream *stream) {
    int open_paren = 0, close_paren = 0, count = 0;
    Atom atom = evaluate(stream);
    string expression = atom.value;
    expression = expression.substr(1, expression.size() - 2);
    string remainder = "";
    do
    {
        if (expression.at(count) == LPAREN) open_paren++;
        else if (expression.at(count) == RPAREN) close_paren++;
        count++;
    } while (open_paren != close_paren);
    remainder = "(" + expression.substr(count + 1, expression.size() - 1);
    return Atom(LITERAL, remainder);
}
```

12.2 Class Topics

The way car and cdr work was based on the examples from the Lisp lecture in class.

12.3 Additional Implementation Details & Screenshots

```
>(car '(1 2 3 4))
1
>(cdr '(1 2 3 4))
(2 3 4)
>(car (cdr '(1 2 3 4)))
2
```

13 THE BUILT-IN FUNCTION: "CONS"

13.1 Implementation

This function was the hardest to get working properly for me. However, I eventually found a way to get it working by manually building the resulting structure in a for loop. I first evaluate the two operands for cons, in case one of them is a variable, then I construct the new list and return it.

```
Atom Parser::cons(InputStream *stream) {
    Atom first_element = evaluate(stream);
    Atom second_element = evaluate(stream);
    string new_list = "(";
    new_list = new_list + first_element.value + " ";
    for (size_t i = 1; i < second_element.value.size(); i++) {
        new_list = new_list + second_element.value.at(i);
    }
    return Atom(LITERAL, new_list);
}
```

13.2 Class Topics

The behavior of cons was taken from the examples of cons in the Lisp lecture in class.

13.3 Additional Implementation Details & Screenshots

```
>(cons 'a '(1 2 3 4))
(a 1 2 3 4)
>(cons '(3 4 5) '(1 2 3 4))
((3 4 5) 1 2 3 4)
>(define x 'a)
x
>(cons x '(1 2 3 4))
(a 1 2 3 4)
```

```
>(cons 'to '(be or not))
(to be or not)
```

This function relies heavily on the quote section of the lexer to tokenize the input string quotes correctly.

14 SQRT, EXP

14.1 Implementation

These functions use the <cmath> C++ standard library to calculate the correct output. The cmath library was useful for efficiently and quickly calculating the square root and exponentiation of the inputs.

```
Atom Parser::my_sqrt(InputStream *stream) {
    int operand = stoi(evaluate(stream).value);
    return Atom(NUMBER, to_string(sqrt(operand)));
}

Atom Parser::my_pow(InputStream *stream) {
    int base = stoi(evaluate(stream).value);
    int power = stoi(evaluate(stream).value);
    return Atom(NUMBER, to_string(pow(base, power)));
}
```

14.2 Class Topics

I used the C++ libraries since the project description allowed us to use libraries as long as we listed why and how we used it.

14.3 Additional Implementation Details & Screenshots

```
>(pow (+ 2 2) (+ 4 4))
65536.000000
>(sqrt (+ 2 2))
2.000000
```

The cmath functions return a double, so the output is calculated to several decimal places of precision.

15 >, <, ==, <=, >=, !=

15.1 Implementation

The implementation of the boolean operators was done in the eval_boolean method in the Parser class. It contains a big switch statement that will do different things depending on which operation is being performed.

```
Atom Parser::eval_boolean(Atom atom, InputStream *stream) {
    switch (atom.token_val) {
        case LESS_THAN: {
            int operand1 = stoi(evaluate(stream).value);
            int operand2 = stoi(evaluate(stream).value);
            bool result = (operand1 < operand2);
            if (result == true) {
                return Atom(TRUE);
            } else {
                return Atom(NIL);
            }
            break;
        }
        case LESS_THAN_EQUAL: {
            int operand1 = stoi(evaluate(stream).value);
            int operand2 = stoi(evaluate(stream).value);
            bool result = (operand1 <= operand2);
            if (result == true) {
                return Atom(TRUE);
            } else {
                return Atom(NIL);
            }
            break;
        }
        case GREATER_THAN: {
            int operand1 = stoi(evaluate(stream).value);
            int operand2 = stoi(evaluate(stream).value);
            bool result = (operand1 > operand2);
            if (result == true) {
                return Atom(TRUE);
            } else {
                return Atom(NIL);
            }
            break;
        }
        case GREATER_THAN_EQUAL: {
```

15.2 Class Topics

The pre-fix notation that we covered in class made implementing this method easier than if we were using in-fix notation. This is because pre-fix notation lends itself better to recursive methods, which Lisp uses heavily. I also used NIL to represent false since that is how Lisp does it.

15.3 Additional Implementation Details & Screenshots

```
>(<= 5 10)
T
>(<= 10 5)
NIL
>(< 100 2)
NIL
>(== 100 100)
T
>(!= 50 1)
T
>(>= (+ 100 100) 2)
T
>(!= 100 100)
NIL
```

The implementation of this part was very similar to the implementation of the arithmetic evaluation part. In fact, the boolean evaluation supports arithmetic evaluation as part of its expression, as shown in the second to last example.

16 FINAL THOUGHTS

This project was definitely challenging. I had to learn a lot about Lisp and declarative programming in general. It showed me a lot of Lisp's strengths that I wouldn't have noticed otherwise, such as the power of its recursive functions. or the list operation functions like car and cdr. Having to write an interpreter to execute Lisp code forced me to really study and understand how Lisp code is supposed to work and how it should be written. Overall, I learned a lot about Lisp and gained some appreciation for it that I didn't have before.

I think choosing to code it in C++ added additional complexity that wasn't necessary but I found ways to get around the complexity by using classes and objects to simplify the operations.

These are the C++ libraries that I used for the interpreter.

```cpp
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <stack>
#include <unordered_map>
#include <cmath>
#include <stdio.h>
#include <regex>
#include <fstream>
#include <limits>
```

```cpp
enum Tokens {
    PLUS = '+',
    MINUS = '-',
    MULTIPLY = '*',
    DIVIDE = '/',
    LPAREN = '(',
    RPAREN = ')',
    QUOTE = '\'',
    SPACE = ' ',
    TRUE = 'T',
    NIL = '~',
    LESS_THAN = '<',
    LESS_THAN_EQUAL = '}',
    GREATER_THAN = '>',
    GREATER_THAN_EQUAL = '{',
    EQUAL = '=',
    NOT_EQUAL = 'N',
    NOT = '!',
    AND = '&',
    OR = '|',
    IF = 'I',
    EMPTY = '_',
    ERROR = '%',
    UNDEFINED = '$',
    NUMBER = 1,
    SYMBOL = 2,
    LITERAL = 3,
    FUNCTION = 4,
    TOKEN = 0
};
```

These categories really helped me to keep track of what each Atom was and how I should process it during runtime.

16.1 Reflection

While my implementation doesn't cover all possible scenarios and there are definitely improvements to be made, I think overall I did a good job of keeping the code relatively clean and organized. I did this by separating the code into different objects and classes as well as separating the tokens into different categories.