Senior Project

A Common Lisp Interpreter in JavaScript

Kellie Dinh
kdinh@brynmawr.edu

Advisor: Richard Eisenberg

Bryn Mawr College, Bryn Mawr PA
Bachelor of Arts in Computer Science

Abstract

Building an interpreter for a programming language is a complex task. Given the programming language Lisp, this project focuses on building an interactive interpreter for Lisp in JavaScript. Background on the syntax and semantics of Lisp is presented to preface the technical details of the interpreter. Each step of the reading, evaluation, and printing stages is outlined as well as prominent features like file loading. The paper discusses some related works and reflects on the experience of developing this large-scale programming project.

1 Introduction

This project describes an interpreter for the Common Lisp programming language written in JavaScript. The interpreter is capable of evaluating simple expressions and calculations and also supports extending the environment through defining new variables. The interpreter, called klisp, uses GNU's CLISP heavily as a reference implementation. This paper details the motivation, background, and approach of this project in order to understand the overall program structure of the interpreter.

Common Lisp is a programming language known for its flexibility and simplicity encompassing many different common programming paradigms (procedural, functional, object-oriented) [7]. These paradigms make up how computer science is taught and practiced today and were first implemented in Lisp [4]. By learning how Common Lisp is interpreted and evaluated, we can see fundamental language design features that power much of the software built today.

2 Background

In order to understand the approach to building a Common Lisp interpreter, some background information on the runtime environment, syntax, and semantics of Lisp will be presented.

2.1 Runtime Environment

Lisp is evaluated in an interactive system called the toplevel [4]. This programming environment consists of a read-eval-print loop (REPL) that takes Lisp expressions, evaluates them, and prints the result. Each part of a REPL refers to a function or set of functions that make up an interpreter. The read function takes in user input and parses its contents into a tree or list data structure. This data structure is the intermediate structure that makes the expressions able to be evaluated. Parsing consists of separating the input into atoms, or individual elements that have lexical meaning in the programming language. These atoms are then evaluated to their final value; this may be simply returning itself (in the case of numbers) or looking up its value in a symbol table (in the case of variables). Once each atom has been evaluated, the final output is printed and then the loop begins again with reading input.

2.2 Parenthetical Syntax

In Lisp, expressions are read left to right and are either a singular atom or a list. An atom is a singular entity that can make up a list and can not be further divided. Examples of atoms include numbers, variables, symbols, and strings (although *klisp* does not support strings or characters). A list is an expression made up of zero or more atoms [2]. Lists are enclosed within pairs of parentheses to indicate where the expression begins and ends. We will see later how the Lisp's parenthetical syntax aids us in parsing Lisp expressions.

Expressions are written as lists in Lisp using prefix notation. The first element in the list is a symbol that represents either a function, a macro (user-defined extension of the language), or a lambda. The remaining elements in the list are the arguments. An argument can be another list or an atom. An example of a simple arithmetic expression in Lisp is written as:

```
(+ 1 2)
```

Using this structure of a list with the first element being the symbol and the following elements being the arguments, we can write statements with arithmetic and truth operators as well as extend the environment by defining new variables.

The `setq` function is used to assign values to new variables in Lisp. It takes in two arguments, where the first argument is the symbol to be assigned and the second argument is the symbol's new value. This example sets value of x to the number 5.

```
(setq x 5)
```

The basic form of a Lisp expression translates to more complex forms. We can use expressions to represent list data structures, definitions of new functions, and control abstractions like loops. Aside from a few special forms of expressions, most expressions follow the list expression form outlined above. Below is a comparison

of an `if` statement that returns `T` (true) if the number inputted is positive and `NIL` (false) if the number is negative in Lisp and Java.

Listing 1: Lisp

```
(if (> x 0)
  T NIL)
```

Listing 2: Java

```
if (x > 0) {
  return true;
}else{
  return false;
}
```

The differences between the two languages are most apparent in their syntactic forms. Lisp syntax contains expressions that are either a list or an atom and the forms usually follow the list expression form mentioned previously, unless there is a keyword. Java syntax is much more complex, containing exponentially more keywords and syntactic forms than Lisp [6]. Writing an interpreter for Java would be much longer and more complex due to the number of syntactic forms the interpreter would have to handle.

2.3 Defining Functions

Defining new functions in Lisp can be done by using the function `defun`. This function takes in multiple arguments. The first argument is the symbol to be assigned a function. The second is a list of the function's parameters. The rest of the arguments are the body of the function - expressions that will be run when the function is called to produce the return value. Here is how a function that increments the given value by 1 might be defined:

Listing 3: Lisp

```
(defun add1 (x)
  (+ 1 x)
)
```

Listing 4: Java

```
int add1 (int x) {
  return 1 + x;
}
```

In Lisp, a symbol can refer to a function, variable, or object. In the case of objects that are lists, they are treated as atoms and not as expressions with the first element being the symbol and the remaning elements being the arguments. In order to process these lists as objects, they must be quoted so that they are not interpreted as expressions. Quoting is done using the function `quote` or the symbol `'`. This example shows the setting of a variable named `nums` to the value of a list containing the numbers 1, 2, and 3.

```
(setq nums '(1 2 3))
```

Understanding Lisp's syntax and semantics is important to building a robust interpreter. With the brief overview of the language presented above, we are able to implement a large portion of Lisp's core functionality and features.

3 Approach

I chose to implement the interpreter in JavaScript because of its ability to write in both functional and object-oriented programming styles. JavaScript is untyped and its arrays are dynamic so the size and the data types in the array do not have to be specified. I find these language features to be forgiving and as Lisp is also an untyped language, implementing its features in another untyped language provides less restriction in implementation.

To run a JavaScript program via the command line you can use Node.js by running `node filename.js` and replacing `filename.js` with the name of your JavaScript file. If the computer does not already have Node.js installed then it can be installed via Homebrew [1] on Mac and Linux machines[2] and via the Node Installer for Windows [3]. To run *klisp* in the command line, input `node repl.js`. Using Node.js also allows for the ability to install JavaScript packages easily using npm [4]. I will use Chai [5] as the testing framework which allow assertions similar to Java's JUnit[6] testing to compare expected behavior and output. The testing framework can be installed once in the Node.js terminal by running `npm install chai`. Once that is installed, the set of tests for *klisp* can be run by inputting the command `npm run test`.

4 Structure of Program

The main program structure for *klisp* mirrors the general structure of an interpreter with steps that correspond to each part of a read-eval-print loop. Reading input from the user is converted to a string, where it is then parsed into tokens and then categorized into JavaScript objects I will refer to as *atoms*. These atoms in JavaScript are different from the atoms in Lisp. Each JavaScript atom has two fields: *type* a string representing the data type and *value* which can be either a string, number, or another atom. These atoms are then added to the abstract syntax tree (AST) which is implemented as a JavaScript array. The AST is then passed to the evaluator which finds the appropriate function in order to evaluate the contents of the AST. The result of evaluation is another AST that is then passed to a print function that formats the output to be printed to the user. After printing, the loop begins again waiting to accept user input. The only way to exit the program is by inputting `(quit)`.

[1] https://brew.sh/
[2] https://docs.brew.sh/Homebrew-on-Linux
[3] https://nodejs.org/en/download/
[4] https://www.npmjs.com/
[5] https://www.chaijs.com/
[6] https://junit.org/junit5/

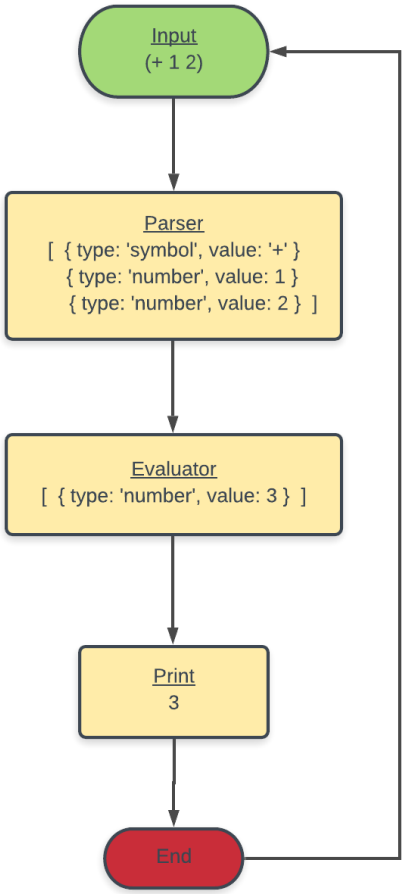

Figure 1: Flowchart showing the high-level process of the interpreter handling the input `(+ 1 2)`.

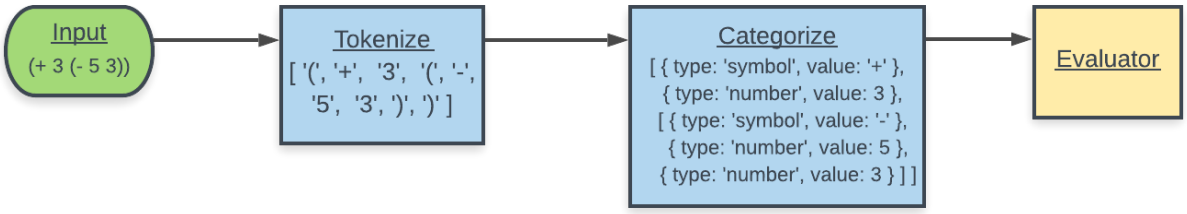

Figure 2: Flowchart showing the intermediary processes involved in parsing the input `(+ 3 (- 5 3))`.

4.1 Reading and Parsing

In order to read in data from standard input, I use the Node.js module *readline*[7]. The *readline* takes user input line by line and converts them to a string. The string is trimmed for whitespace at the beginning and end and passed to the tokenizer. The tokenizer takes a string and puts spaces around each parenthesis and splits on whitespace. The tokenizer is based on the `tokenize` function in Cook's *Little Lisp Interpreter*[3]. For example, it takes the Lisp input of `(+ 3(- 5 3))` and transforms it to `( + 3 ( - 5 3 ) )` and transforms that to `[ '(', '+', '3', '(', '-', '5', '3', ')', ')' ]`. One difference I added to `klisp`'s tokenizer was the ability to handle quoted elements. Punctuation next to a number such as `3+` was tokenized as 3 which is desirable, yet this translated to quoted elements like `'b` to be tokenized without the quote. The `tokenize` function in *klisp* includes the manual checking for quotes in the token and separates the quote from the element if the two are joined in one token.

The next phase in parsing is categorizing. The main functions involved in this phase are `parenthesize` and `categorize`. These functions create the AST structure to be passed on to the evaluator and printer. The AST is made up of JavaScript arrays that contain atoms. For each token received, there are two ways that the `parenthesize` function will handle the token. If the token is an opening parenthesis a new list is created. If the token is a closing parenthesis the current list is popped off and added to the AST. If the token is not a parenthesis,`categorize` is called to create the atom object which is then added to the AST. `Parenthesize` is called recursively on each token present.

Creating the atoms to be added to the AST is performed in the function `categorize`. The function takes in a token and categorizes it as either a number or symbol. The type is set as a string literal i.e. `type: 'symbol'`. The value is set to the actual token that is passed in i.e. `value: 'x'`. Special handling for quote was implemented so that the `'` symbol was given the value of the word `quote` rather than the punctuation mark. Once we have the AST representation of the Lisp input, we can begin evaluation.

4.2 Evaluation

Evaluation is at the core of the interpreter and is the most complex process in the program. The evaluation process takes in an AST representation of the Lisp input, evaluates its contents based on the syntax and semantics of Lisp, and outputs the final value in the same AST form to be passed on to the printing stage. The process of evaluation involves many functions, the first one being `output`. `Output` iterates through each element in the AST and determines which function to pass the AST to based on the type of that element. Aside from handling the special case of `quit` which exits the program, the main decision is whether the element is a singular atom or a list. We can check if an element is a list by using JavaScript's `instanceof` operator to check whether the input belongs to the `Array` class in JavaScript. If the element is a list, we pass the AST to the `evaluateList` function. If the element is a singular atom, we pass the AST to the `evaluate` function.

[7] https://nodejs.org/api/readline.html

Lisp input	AST representation to `output` function
5	[type: 'number', value: 5]
(+ 3 2)	[type: 'symbol', value: '+' , type: 'number', value: 3 , type: 'number', value: 2]
(cons (+ 3 3) '(A))	[type: 'symbol', value: 'cons' , [type: 'symbol', value: '+' , type: 'number', value: 3 , type: 'number', value: 3], [type: 'symbol', value: 'quote' , type: 'symbol', value: 'A']]

Table 1: Table showing Lisp input and its AST before evaluation

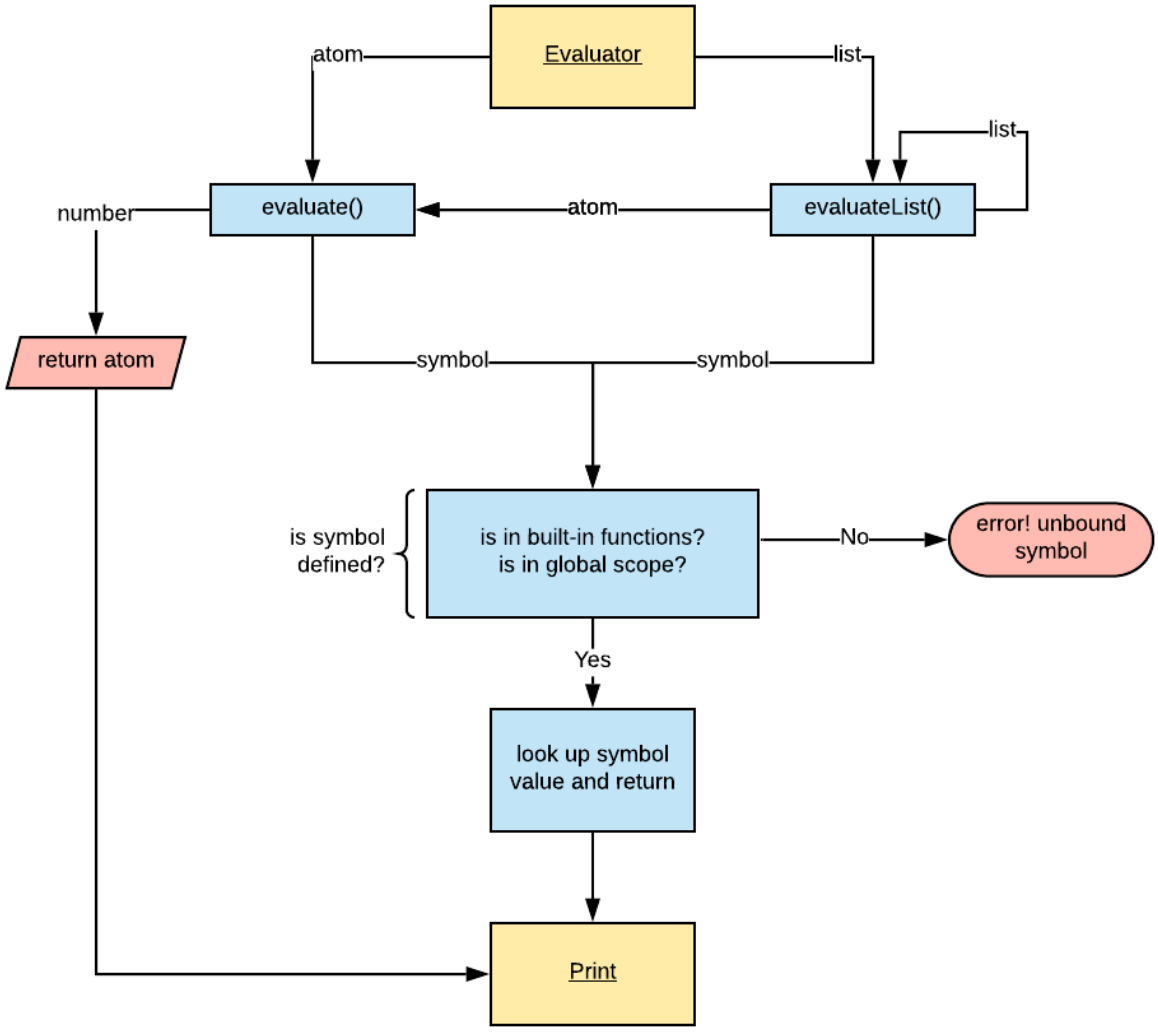

Figure 3: Flowchart showing the process of the interpreter evaluating the AST

Evaluating an atom

Evaluation of an atom is performed in the `evaluate` function. The function expects the input to be a singular atom and uses the `type` and `value` fields on the input. Based on the atom's type and value, we can determine how to process the atom. The simplest evaluation is for numerical atoms. If the atom's type is a number, then we can simply return the whole atom. For symbols, there are two cases: either the atom is a built-in function that was called without arguments or the atom is a user-defined variable in the global scope. If the atom is a built-in function, we return that function call. If the atom is a user-defined variable in the global scope, we look up that variable in the global symbol table for variables (named `global_vars`) and return the whole atom.

Evaluating a list

Evaluating a list proves to be a more complex procedure due to the wider range of types of inputs. Lists encompass the wide range of built-in functions that can be called in `klisp` as well as user-defined functions in the global scope. Lists can also be nested inside of other lists. Besides lists and symbols, the other format for input to `evaluateList` can be an atom as the argument to a function which we can pass to `evaluate` to extract the value from.

In order to find the corresponding function for a symbol, we make use of JavaScript's notion of function expressions to group together multiple function definitions with similar usages. Having functions defined as expressions and grouped under one variable allows for easier readability and access. For example, the arithmetic operators (`+, -, *, /`) and truth operators (`>, <, >=, <=, =, /=`) are grouped in the variable `operators`. We use the `in` keyword to check whether the symbol is in the `operators` variable of functions. Other sets of functions are grouped in the variables `list_operators`, `special`, and `print_functions`.

For functions that are not in the variables representing built-in functions in Lisp, we want to check whether it is a user-defined function. User-defined functions are created using the Lisp function `defun` and are added to the global symbol table for functions (named `global_functions`). The lookup process to check whether a given symbol corresponds to a user-defined function simply consists of a linear search through the `scope` array in `global_functions`.

Algorithm 1 Evaluating a list

```
Require: Input is AST representation of Lisp input initialize context
  for all elements in AST do
    if element.type is 'symbol' then
      if element is a defined function then
        set element as InnerContext.functions_in_context
      else
        return error!  - unbound symbol
      end if
    else if element is an instance of an Array then
      evaluateList(element)
    else
      evaluate(element)
    end if
  end for
```

If the input is an array, we pass the arguments to the function evaluateList. In this function, we begin by initializing a context object that will hold the functions and variable scopes. We iterate through each element in the input and process the list based on the first element in the list.

If the first element is a symbol, we check if it is any one of the predefined symbol functions in Lisp (arithmetic, truth, conditional). If it is one of the predefined symbol functions, we call that function with the rest of the elements as its arguments, or store its value in `InnerContext.function_in_context` if we are not at the end of the list in order to continue evaluation. If it is not one of the predefined symbol functions, we attempt to look up its value in the global symbol table, `global_functions`.

Other cases are if the element is another list, we call `evaluateList` again on that list. If the element is not a symbol or another list, we call `evaluate` on that element.

After evaluating each element, we push its evaluated result to the `InnerContext` object created at the start of the function. If there was a function stored previously in the context, we call that function with the elements in that context's scope. The result of that function call is returned as the final value to be passed into the `print` function.

4.3 Representing Variables

Two types of classes are defined in *klisp*, the `InnerContext` class and `GlobalContext` class in order to keep track of variables. The `InnerContext` class is used within the `evaluateList` function to implement block scoping and the `GlobalContext` class is used to implement a global symbol table for variables and function definitions. This section focuses on the `InnerContext` class and how nested expressions are evaluated and how intermediary values are kept track of.

Inner Variable Scopes

Block scoping in `klisp` keeps track of intermediate values returned through nested function calls and function definitions to be called once its arguments are evaluated.
Variable scopes are implemented as JavaScript classes with two main fields: an Array named `scope` and a variable named `function_in_context`. The `InnerContext` class is only used within the `evaluateList` function. Since lists contain a symbol as its first element, we save that symbol's corresponding function as the `function_in_context`. When there is a nested function call to either `evaluate` or `evaluateList`, rather than returning the output atom we add it to the `scope` array. At the end of `evaluateList`, we call the function set as the `function_in_context` with the elements in the `scope`. The result is saved in the variable `new_args` which is then returned.

4.4 Global Variables and Functions

Each user-defined variable and function is saved in a global symbol table. The symbol table is implemented as a class in JavaScript called `GlobalContext` which contains the two Arrays `scope` and `parent`. The `parent` field was added for further implementation of local variables and nested scopes but is not utilized in this version of `klisp`.

User-defined variables are created using the function `setq`. The value can be either a number or a list. After evaluating the value (which is done when the value is an expression), we create an atom and set the `type` field to be the variable name, and the `value` field to be the atom representation of the value. For example, we would create the following atom after calling `(setq x 5)`:
`{ type: 'x', value: { type: 'number', value: 5 } }`.

Keeping the atom representation rather than the literal value for the `value` field proves to be more consistent with how lists are stored. Since lists are stored as arrays of atoms, we can set the `value` of the variable as the entire array. After calling `(setq nums '(1 2 3))`, we create the following atom:
`{ type: 'nums', value: [ { type: 'number', value: 1 }, { type: 'number', value: 2 }, { type: 'number', value: 3 } ] }`

User-defined functions are created using the function `defun`. The arguments to `defun` are then placed in a newly-created function object that contains four fields: `type`, `name`, `parameters`, and `body`. The `type` is set to the string `'function'` and the `name`, `parameters`, and `body` arguments are set to the first, second, and third arguments to `defun` respectively.

Once we have the object representation of the global variable or function, we can add it to the global symbol table. The functions `get_var` and `get_func` both perform a linear search and returns the variable object if one already exists with the same name. If one does not already exist with the same name, the functions return `null`. If `get_var` or `get_func` returns null, we can simply add the object representation of the global variable or function to its respective symbol table, either the `scope` field of `global_var` or `global_functions`. If `get_var` or `get_func` does not return null, we know that there exists an object with the same name so we replace the existing object with the new object.

Figure 4: Chart showing how the nested expression `(+ 3 (- 5 3))` is evaluated by iterating through each element and creating two `InnerContext` objects, one for the inner `(- 5 3)` expression and one for the outer expression that contains 3 and the result of `(- 5 3)` which is 2. The inner expression's context object is written in bold and is bordered with a heavier line weight.

Element	Atom	Procedure	InnerContext
+	{ type: 'symbol', value: '+' }	set '+' as InnerContext.functions_in_context	{ scope: [ ], function_in_context: [Function: +] }
3	{ type: 'number', value: 3 }	- call **`evaluate`**({ type: 'number', value: 3 }) - add { type: 'number', value: 3 } to InnerContext.scope	{ scope: [{ type: 'number', value: 3 }], function_in_context: [Function: +] }
(- 5 3)	[{ type: 'symbol', value: '-' } { type: 'number', value: 5 } { type: 'number', value: 3 }]	- call **`evaluateList`** with the list of atoms - new InnerContext object is created **(bolded)**	**{ scope: [ ], function_in_context: undefined }**
-	{ type: 'symbol', value: '-' }	set '-' as InnerContext.functions_in_context	**{ scope: [ ], function_in_context: [Function: -] }**
5	{ type: 'number', value: 5 }	- call **`evaluate`**({ type: 'number', value: 5}) - add { type: 'number', value: 5 } to **InnerContext.scope**	**{ scope: [{ type: 'number', value: 5 }], function_in_context: [Function: -] }**
3	{ type: 'number', value: 3 }	- call **`evaluate`**({ type: 'number', value: 3}) - add { type: 'number', value: 3 } to **InnerContext.scope**	**{ scope: [{ type: 'number', value: 5 }, { type: 'number', value: 3 }], function_in_context: [Function: -] }**
		- call function_in_context: [Function: -] with **InnerContext.scope** as the arguments - add { type: 'number', value: 2 } to InnerContext.scope	{ scope: [{ type: 'number', value: 3 } , { type: 'number', value: 2 }], function_in_context: [Function: +] }
		- call function_in_context: [Function: +] with InnerContext.scope as the arguments **return { type: 'number', value: 5 }**	

4.5 Error Handling

In *klisp*, errors are implemented as a Node.js module defined in the file `error.js`. Errors in *klisp* are derived from JavaScript's `Error` class. These classes contain a constructor function that takes in a string parameter named `message` that is by default blank. The custom-defined errors are meant to be broad and are not all-encompassing. They are `NumberError`, `InputError`, `NumArgsError`, `DivideZeroError`, `ListArgError`, `UnboundSymbolError`, and `UnboundFunctionError`. The module also contains some sample error messages that are implemented as functions that return the string containing the message. For example, the function`num_args_error` returns the string, "`incorrect number of arguments`". These simple functions can be used with each other to build more complex error messages.

When a custom-defined error is thrown, it is passed to the `handle_errors` function. This function builds the string error message to be printed to the user based on the type of error thrown and whether a string was passed in as the `message` parameter. Defining only a few custom errors pertaining to broad types of errors rather than meticulously defining a new class for every type of error proves to be beneficial to both the user and the programmer. Let's illustrate how we can use the wide-ranging `InputError` to convey different types of error messages to the user. For example, the `setq` function in Lisp expects the first argument to be a symbol. If the user inputs a number or a list as the first argument, we want to throw an error because of the input's incompatible type with the function. Similarly, in `output` when evaluating lists, we expect the first element of the list to be a symbol. While these two types of errors are similar in that they pertain to incorrect type of input, they differ in the format as `output` is iterating through a list while `setq` is iterating through individual atoms so grouping them with the same error output i.e. `"Incorrect format of input"` is a solution albeit vague and not very helpful to the user.

Rather than define two separate errors that pertain to incorrect input we can throw one type of error, `InputError`, where the message can be altered to match the use case.

Listing 5: Use of InputError in function `setq` (klisp.js:649)

```
if (variable.type !== 'symbol') {
      throw new error.NumArgsError("first argument must be a symbol - ");
}
```

Listing 6: Use of InputError in function `output` (klisp.js:928)

```
if (input[0].type !== 'symbol' && input.length > 1) {
      throw new error.InputError("Unexpected head of list: " +
                                 (input[0]).value + " - ");
}
```

After an error is thrown, it is caught and passed to `handle_errors` which defines the default error output for each type. For every `InputError`, the function `unknown_error` is called which appends the string `"input error"` to the output. Since the error was thrown with a string value for `message`, that string is appended to the final output. So when a user attempts to call the `setq` with a number or list as the first argument:

```
error! first argument must be a symbol - input error
```

When a user enters inputs an expression with the first element as a number or list and not a symbol, i.e. `(1 2 3)` they receive the following error output:

```
error! Unexpected heaf of list: 1 - input error
```

Implementing errors with flexibility in the construction of output messages allows the programmer to tailor the error messages to their preference. Many of the error messages were based on GNU's CLISP implementation[8]. Since *klisp* does not quit because of errors, it is important to make sure each error is accounted for properly in the `catch` clause. With this outline of the structure of *klisp*, we can now go into deeper detail about some of the key functions that power this interpreter.

[8] https://clisp.sourceforge.io

5 Key Functions

Each function in *klisp* is implemented in the global scope and has a simple function signature usually containing one parameter. In each function, it is assumed that the input is of the correct type or it throws an error.

5.1 Parse

The `parse` function is the first function that is applied to the Lisp input and is called in the file `repl.js`. It is the application of the `tokenize` and `categorize` functions. The function takes in the user-entered string from standard input and outputs either a JavaScript array representing the AST or a JavaScript object representing a singular atom in Lisp.

5.1.1 Tokenize

The `tokenize` function takes in a string of Lisp input and outputs an array of individual tokens separated by whitespace. For handling quoted symbols like `'a`, we replace cases of the `'` character with the word `quote`. For handling quoted symbols with the word like `quote a` that are incorrectly tokenized as `quotea`, we manually check whether the token contains the substring `quote` at the beginning indices and if the length is greater than 5 (the number of characters in `quote`). If this condition is true, then we split the string into two tokens: one containing the word `quote` and the other containing every other character after that which is the symbol name.

5.1.2 Parenthesize

The `parenthesize` function takes two parameters named `input` and `list`. This function takes a list of tokens as the input and returns the completed AST. It recursively iterates through each token in the list and calls `categorize` on each one, unless it is an opening or closing parenthesis which are used to group nested expressions in their own array by creating a new array or popping off the current one.

5.1.3 Categorize

The `categorize` function takes in a singular string token and returns a JavaScript object that is the atom representation of that token. There are three cases for this function: whether the input is a `number`, `quote`, or a `symbol`. The objects returned from this function are used to create the AST.

5.2 Output

The function `output` performs the major evaluation in the interpreter and is called after `parse` in `repl.js`. It expects the JavaScript array representation of the AST and outputs either another AST or a JavaScript object representing a singular atom in Lisp. This function iterates through each element in the input AST and deduces whether to pass the element to `evaluate` or `evaluateList`. For special behavior like `quit`, the string literal "quit" is returned. The final evaluated AST is passed to the `print` function.

5.3 Print

Once we have the evaluated AST, we must rebuild the Lisp representation based on this implementation in order to print the output. `Print` is one of the more sophisticated functions and handles a wide range of inputs to be printed. The nested structure of the AST structure using JavaScript arrays makes the general printing rules quite simple: print each element's value and if at the beginning of the list append an open parenthesis; if in the middle of the list append a space; if at the end of the list append a closed parenthesis. For handling of `quote`, we need to keep track of when it appears. Since parsing doesn't nest the `quote` symbol with its arguments but rather puts the `quote` symbol on the same level as its arguments, we need to keep track of when `quote` appears in order to flag the `closed_paren` variable. When `print` is called with

the `closed_paren` flag set to true, we append a closed parenthesis at the end of the output string. Also, symbols containing alphabetic characters are capitalized at print time.

5.4 Setup

`Setup` is a simple function called at the start of the program before accepting any user input. It adds the two truth values, `T` and `NIL` to the global symbol table `global_var.scope` so that they can be accessed the same way user-defined variables are but that are built-in to the interpreter.

5.5 Load

The `load` function is a Lisp command that reads in input from a file on the user's computer rather than through standard input and is handled by the function `fileReader` in `repl.js`. If the user inputs the keyword `load` at the beginning (string indices 1 through 5), the `fileReader` function is called with the remainder of the input (string indices 7 through input.length - 1). The function then calls the Node.js function `readFileSync`[9] that searches for and opens the file and reads each line of input into the variable `lines`. Then, each line of input is passed through *klisp's* `output` function to be evaluated. At the end of evaluation if no errors are thrown, we return the value `T` on success.

6 Discussion

Some key features of this interpreter are the ability to handle nested expressions and define new variables and functions using `setq` and `defun` in the global scope. It can handle defining complex nested functions and recursive functions like computing the nth factorial number. Currently, `klisp` does not support local variables or closures using `let` or `do`. This interpreter also does not handle some commands like `progn`, `function`, `apply`, and `eval`. This implementation consists of three main files, `klisp.js`, `repl.js`, and `error.js` which total to 1280 lines of code. Compared to another Lisp interpreter written in JavaScript, Cook's *Little Lisp Interpreter* contains 116 lines of code [3]. Although `klisp` provides some more functionality namely in handling nested expressions, it isn't the most efficiently written program with some repetitive code structures.

7 Related Work

This project would not be possible without the aid of these resources.

7.1 CLISP

CLISP is an implementation of Common Lisp released by the GNU Project and is free and open-sourced [8]. The full implementation contains an interpreter which was used as a reference throughout the development of this project. Debugging strange output, handling types of input errors, and all aspects of general program behavior were based on CLISP's interpreter.

7.2 Little Lisp Interpreter

Mary Rose Cook's *Little Lisp Interpreter* [3] was referenced throughout the development of this project for its simplicity, program structure, and use of modern JavaScript features. The project is a Lisp interpreter written in JavaScript that gained popularity in the modern JavaScript developer community for its light weight (at 116 lines of code) and simplicity in design. I was inspired by *Little Lisp's* program structure and implemented concepts like defining a context class, creating the program as a Node.js module, and defining the functions as expressions directly in my own program. Some functions are direct replicas of functions in *Little Lisp* only with some added cases to account for the wider range of commands that *klisp* supports. Major decision decisions like separating evaluaton based on whether the input is an atom or a

[9] https://nodejs.org/api/fs.html

list, implementing the AST as nested JavaScript arrays, and constructing an atom with the fields `type` and `value` were directly taken from *Little Lisp*, so I am very grateful for its existence and the accompanying blog post[5] that breaks down the program structure.

7.3 Lisp Dialects

The Lisp programming language has been in existence since the 1950s and the dialect that this project focuses on, Common Lisp, is only one of many dialects and subsects that have spawned from the original Lisp. One dialect is Clojure[10], which shares many of the core programming language design decisions as the original Lisp. Clojure is able to interact with Java and JavaScript programs and libraries so it is widely used in the technology industry today. Another dialect of Lisp is Racket[11] which is a a popular language for Computer Science education. What is remarkable about the existence of these dialects is that they still strongly adhere to the language features introduced in the original Lisp language. Concepts like the REPL, if/then/else conditional form, and recursive function calls are elements of Computer Science that we are now accustomed to in our software. The persistence of these programming paradigms that are present in modern-day technologies shows the lasting impact of Lisp and gives context for where this project fits in a Computer Science education.

8 Reflection

This project was a great exercise in strengthening my understanding of Lisp and JavaScript. Some of the major takeaways from this experience were gaining a deeper knowledge of JavaScript, testing frameworks, and Software Engineering best practices like project management issue tracking.

I chose to undertake this project in JavaScript, a language that I was somewhat familiar with but did not have much formal programming experience in because it is widely used in the Software Engineering industry and I wanted to gain more experience working on large-scale projects in JavaScript. Some of the major concepts I learned throughout this experience were language features like the difference in variable declarations using `let` and `var` as one is used for global scoping and the other for local scoping. Other valuable JavaScript features I learned from this experience are that because the language is untyped, it is possible to add any type of object to a list. This feature proved to be beneficial in implementing variable scopes as the type of input can vary from a single JavaScript object to an Array so specifying the type would be tedious. Also, referencing *Little Lisp* introduced me to using function definitions as variables which helped in organizing functions based on usage such as `operators`, `list_operators`, etc. Organizing functions this way greatly improved the readability of the program which is one of it's main advantages. Also, I am proud of integrating a testing framework I was previously unfamiliar with into this project and the breadth of tests written for it. Once I integrated testing into my workflow, development on this project became exponentially smoother as one command could test for dozens of errors versus manually inputting test commands which is much more time consuming.

Given more time, there are some aspects of the project that I would want to re-do. In the `parse` function, the handling of the `quote` symbol is very tedious. I would like to update the `tokenize` function to accurately parse quotes and symbol and not merely searching for a substring of the word `'quote'` like the current implementation.

Another process I would like to improve would be that of evaluation. The ideal program would just be calling three main functions at the toplevel: `read`, `evaluate`, and `print`. Structuring the program more rigidly around these three processes would make the program even more readable and would aid in making the program more accessible to outside programmers who want to extend the program with other features. On a similar vein, making each of these processes recursive instead of iterative would likely aid in readability and accessibility and structure better with the implementation of local variables.

One of the highest rules of software development is DRY - don't repeat yourself. Unfortunately, there are some sections of code that are repeated namely in the definitions of functions in `operators`. These sections of code could be refactored and modularized in order to not have multiple repeated portions.

[10] https://clojure.org
[11] https://racket-lang.org

References

[1] *1. Introduction: Why Lisp?* URL: http://www.gigamonkeys.com/book/introduction-why-lisp.html.

[2] Robert J. Chassell. *An introduction to programming in emacs lisp*. Software Foundation, Inc., 2002.

[3] Mary Rose Cook. *Little Lisp Interpreter*. URL: https://maryrosecook.com/blog/post/little-lisp-interpreter.

[4] Paul Graham. *ANSI COMMON LISP*. Prentice Hall, 2009.

[5] *Mary Rose Cook*. URL: https://maryrosecook.com/blog/post/little-lisp-interpreter.

[6] Peter Norvig. *(How to Write a (Lisp) Interpreter (in Python))*. URL: https://norvig.com/lispy.html.

[7] Guy L. Steele. *COMMON LISP: the language*. 2nd. Digital Press, 1990.

[8] *Welcome to CLISP*. URL: https://www.gnu.org/software/clisp/.