TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
Paschimanchal CAMPUS

PROJECT ON
“C-Based LISP Interpreter”

A course Project Submitted to the Department of Electronics and Computer Engineering in partial fulfilment of the requirements for the Practical course on Computer Programming [CT 401]

Submitted to: Department of Electronics and Computer Engineering ,
Paschimanchal Campus
Institute of Engineering, Tribhuvan University
Falgun 2081

Submitted by:
Saksham Adhikari (081BEI032)

Abstract

This project introduces "Backpain Version 0.0.0.1.1," a lightweight Lisp interpreter developed in C, designed to execute Lisp code both interactively and from files across Windows and Unix-like platforms. The interpreter supports core Lisp data types, including numbers, symbols, strings, S-expressions, and Q-expressions, as well as user-defined lambda functions. It provides a collection of built-in functions for essential operations, such as arithmetic (+, -), list manipulation (head, tail), variable management (def), and control flow (if). Key implementation techniques include recursive descent parsing for processing expressions, dynamic memory management for resource efficiency, and environment-based symbol resolution to enable dynamic scoping. While the interpreter lays a robust groundwork for educational exploration of Lisp and interpreter design, it currently lacks advanced features like macros and tail-call optimization and offers only basic input handling on Windows. Nonetheless, it serves as a practical learning tool for understanding Lisp's syntax and semantics and offers a flexible, extensible platform for future development.

Abbreviations and Their Meanings

- **AST** - Abstract Syntax Tree
  A tree representation of the abstract syntactic structure of source code.
- **COW** - Copy-on-Write
  A resource management technique where a copy of a resource is made only when it is modified.
- **errno** - Error Number
  A variable used in C programming to indicate error conditions during function execution.
- **fmt** - Format
  Used in error messages and string formatting operations.
- **func** - Function
  Refers to function names or types in various programming contexts.
- **LASSERT** - Lisp Assertion
  A macro used for runtime checks or assertions in the Lisp interpreter.
- **lenv** - Lisp Environment
  A structure used to manage symbol bindings in the Lisp interpreter.
- **LVAL** - Lisp Value
  A prefix for enum constants that represent different value types in the interpreter.
- **lval** - Lisp Value
  A structure representing a value within the Lisp interpreter.
- **par** - Parent
  Used in the lenv structure to refer to the parent environment.
- **pos** - Position
  Tracks the current position in the input string during parsing.

- **REPL** - Read-Eval-Print Loop
  The interactive prompt where users input code, which is then evaluated and printed.
- **s** - String
  Commonly used in function parameters (e.g., char* s) to denote a string.
- **sexpr** - S-Expression
  A type of Lisp expression, typically a nested list or atomic symbol.
- **sym** - Symbol
  Refers to symbol names or bindings in the Lisp environment.
- **t** - Type
  Used in ltype_name to denote value types.
- **va** - Variable Arguments
  Used in va_list to handle variadic (variable argument) functions.
- **v** - Value
  A common variable name for an lval* pointer, representing a value.
- **x** - Value
  A common variable name for an lval* pointer, often used for results.

Table of Contents

---

1. Introduction

1.1 Background and Problem Statement

The provided code is a Lisp interpreter implemented in C, named "Backpain Version 0.0.0.1.1." Lisp, a family of programming languages known for its symbolic expression processing and functional programming paradigm, serves as the foundation for this project. The interpreter aims to provide a lightweight, interactive environment for executing Lisp code, capable of handling both real-time user input and file-based execution. It supports fundamental Lisp constructs such as S-expressions (evaluated lists), Q-expressions (quoted lists), numbers, symbols, strings, and functions, including user-defined lambda functions.

The problem addressed is the creation of a minimalistic yet functional Lisp interpreter that operates across different platforms (Windows and Unix-like systems). This tool could serve educational purposes, allowing users to explore Lisp's syntax and semantics, or act as a foundation for more complex systems. The project's early version number (0.0.0.1.1) indicates it is a work in progress, with potential for refinement and expansion.

1.2 Objectives

The project's primary objectives are:

- **Parsing**: Develop a parser to interpret Lisp expressions from user input or files.
- **Data Types**: Support essential Lisp data types (numbers, symbols, strings, S-expressions, Q-expressions, and functions).
- **Built-in Functions**: Implement core functions for arithmetic, list manipulation, variable management, and control flow (e.g., if statements).
- **Platform Independence**: Ensure compatibility with Windows and Unix-like systems using conditional compilation.
- **User-Defined Functions**: Enable the creation and evaluation of lambda functions.
- **Stability**: Incorporate basic error handling and memory management to prevent crashes and leaks.

1.3 Features

The interpreter boasts the following features:

- **Interactive Prompt**: A command-line interface ("Backpain> ") for real-time input and execution.
- **File Execution**: Ability to load and run Lisp code from files specified as command-line arguments.
- **Built-in Functions**: Includes +, -, *, /, head, tail, join, eval, if, def, =, list, lambda, load, error, and print.
- **Memory Management**: Utilizes dynamic allocation (malloc, realloc, free) for efficient resource use.

- **String Handling**: Supports escaping and unescaping of special characters in strings.
- **Comparison Operators**: Provides ==, !=, >, <, >=, and <= for logical evaluations.
- **Environment Management**: Uses a lenv structure to manage variable scoping and bindings.

1.4 Limitations

Despite its capabilities, the interpreter has notable limitations:

- **Windows Line Editing**: Lacks advanced features like command history due to reliance on fgets instead of readline.
- **Basic Error Handling**: May not catch all edge cases, risking memory leaks or undefined behavior.
- **Missing Advanced Features**: Does not support macros, tail-call optimization, or other advanced Lisp constructs.
- **Performance**: Repeated memory reallocations could slow down processing of large expressions.
- **Development Stage**: The early version suggests incomplete features or potential instability.

---

2. Problem Analysis

2.1 Understanding the Problem

The central challenge is constructing a Lisp interpreter in C that can parse input, build an abstract syntax tree (AST) using lval structures, evaluate expressions within a lenv environment, and manage memory effectively. This requires handling nested expressions, function applications, and variable scoping while maintaining portability across platforms.

2.2 Input Requirements

- **Interactive Input**: User-entered Lisp expressions via the prompt (e.g., "(+ 1 2)").
- **File Input**: Lisp code from files specified as command-line arguments.
- **Expression Types**: Numbers (e.g., "123"), symbols (e.g., "x"), strings (e.g., ""hello""), S-expressions (e.g., "(+ 1 2)"), and Q-expressions (e.g., "{1 2 3}").

2.3 Output Requirements

- **Evaluation Results**: Printed outcomes of expressions (e.g., "3" for "(+ 1 2)") using lval_println.
- **Error Messages**: Descriptive errors for invalid inputs or runtime issues (e.g., "Error: Division By Zero").
- **Formatted Strings**: Proper handling of escaped characters (e.g., ""hello\nworld"").

2.4 Processing Requirements

- **Parsing**: Transform input strings into lval objects with functions like lval_read.
- **Evaluation**: Process expressions using lval_eval, executing functions via lval_call.
- **Memory Management**: Allocate and deallocate resources for lval and lenv structures dynamically.
- **Environment Handling**: Maintain symbol-value bindings and scoping with lenv.

2.5 Technical Feasibility

The project leverages standard C libraries (stdio.h, stdlib.h, string.h, stdarg.h, errno.h) and the editline library for Unix-like systems, ensuring feasibility. Conditional compilation (#ifdef _WIN32) enables cross-platform support. Manual memory management, while complex, is standard in C and viable with careful implementation.

---

3. Review of the Related Literature & Methodology

This section would typically survey existing Lisp interpreters (e.g., Common Lisp, Scheme) and C programming techniques. The "Backpain" interpreter draws from Lisp’s emphasis on symbolic computation and functional programming. The use of editline mirrors lightweight command-line tools, while memory management reflects C best practices. Compiler design principles (parsing, ASTs, evaluation) also inform the structure, though specific references are omitted here due to the lack of an attached bibliography.

Here are the key methodologies used in the program, a Lisp interpreter written in C, explained in a clear and structured manner:

3.1. Modular Design

- The program is split into distinct modules or functions, each handling a specific task, such as parsing, evaluation, memory management, and built-in operations. This separation enhances maintainability and allows for easy extension of the codebase.

3.2. Recursive Descent Parsing

- The parser employs a recursive descent approach to process nested expressions. It starts with the top-level expression and recursively parses subexpressions, constructing an abstract syntax tree (AST) using lval structures. This method suits Lisp’s parenthesized syntax well.

3.3. Dynamic Memory Management

- Dynamic memory allocation (malloc, realloc, free) is used to manage lval (value) and lenv (environment) structures. This flexibility supports variable-sized data like lists and ensures proper cleanup to avoid memory leaks.

3.4. Environment-based Symbol Resolution

- Symbols are resolved using a chain of environments (lenv), implementing dynamic scoping. The program searches the current environment and its parent environments for a symbol, raising an error if it’s not found. This supports nested function definitions.

3.5. Function Application with Partial Application

- User-defined functions (lambdas) support partial application. If fewer arguments are provided than expected, a new function is returned with the remaining parameters, enabling functional programming techniques.

3.6. Built-in Functions as First-Class Citizens

- Built-in functions are stored as function pointers within lval structures, treated similarly to user-defined functions. This uniformity simplifies evaluation by using the same mechanism for both function types.

3.7. Error Handling with Propagation

- Errors are encapsulated as lval structures of type LVAL_ERR. When an error occurs during parsing or evaluation, it propagates up the call stack for centralized handling, ensuring graceful error reporting.

3.8. String Escaping and Unescaping

- String literals with escape sequences are converted to actual characters during parsing and escaped appropriately when printed. This ensures accurate handling of special characters and multi-line strings.

3.9. Cross-Platform Compatibility

- Conditional compilation (#ifdef _WIN32) addresses platform-specific differences, such as input handling. Windows uses a custom readline function, while Unix-like systems leverage the editline library for features like command history.

3.10. Interactive REPL (Read-Eval-Print Loop)

- An interactive prompt allows users to input expressions, which are read, evaluated, and printed in a continuous loop. This REPL enables real-time interaction with the interpreter.

3.11. File Loading and Execution

- The program can load and execute Lisp code from files provided as command-line arguments. It parses the file into S-expressions and evaluates them sequentially, supporting script execution.

3.12. Macro-based Assertions

- Macros like LASSERT, LASSERT_TYPE, and LASSERT_NUM perform runtime checks on inputs and conditions, generating errors when necessary. This validates program behavior efficiently.

3.13. Copy-on-Write for Environments

- New environments for function calls copy the parent environment and add new bindings. This prevents changes in the child environment from affecting the parent, maintaining proper scoping.

3.14. Tail Call Optimization (Partial)

- The recursive evaluation structure allows for potential tail call optimization, though not fully implemented. This could improve efficiency for deep recursion in S-expressions.

3.15. AST Manipulation

- Functions like lval_pop, lval_take, and lval_join manipulate the AST, enabling extraction and combination of expression parts. This is key for list operations and function application.

3.16. Value Comparison

- A comparison function (lval_eq) handles lval structures, recursively comparing different types and nested data. This supports equality operations in the language.

3.17. Printing with Formatting

- Functions print lval structures in a readable format, handling various types and escaping strings appropriately. This ensures clear output for users.

3.18. Initialization and Cleanup

- The environment is initialized with built-in functions, and memory is cleaned up on exit. This proper resource management prevents leaks and ensures a consistent starting state.

3.19. Command-Line Argument Handling

- The program processes command-line arguments to switch between interactive mode (REPL) and file execution, offering flexible usage options.

3.20. Versioning and Documentation

- Version information ("Backpain Version 0.0.0.1.1") and comments provide basic documentation, aiding in understanding and debugging the code.

Flowchart

4.1 Algorithms

1. Parsing Algorithm (lval_read)

The parsing algorithm transforms a string of Lisp code into an internal representation called an lval (Lisp value) structure. This structure can represent numbers, symbols, strings, S-expressions, or Q-expressions. Here's a detailed step-by-step process:

Steps:

1. **Initialize Position**:
    - Start at the beginning of the input string, tracking the current character position (pos).
2. **Skip Whitespace and Comments**:
    - Check the current character.
    - If it’s a space, tab, newline, or the start of a comment (';'), move pos forward.
    - For comments, skip all characters until a newline or end of string is reached.
    - Repeat until a non-ignored character is found.
3. **Detect End of Input**:

    - If pos reaches the string’s end (e.g., null terminator), check if more input was expected (e.g., inside an expression). If so, return an lval of type LVAL_ERR with a message like "Unexpected end of input".

4. **Parse S-expressions**:
    - If the character is '(':
        - Create a new lval of type LVAL_SEXPR.
        - Advance pos past '('.
        - Call a helper function lval_read_expr with:
            - The current string.
            - Updated pos.
            - End character ')'.
        - lval_read_expr:
            - While pos doesn’t point to ')', recursively call lval_read for each sub-expression.
            - Add each parsed lval to the S-expression’s list of children.
            - Skip whitespace/comments between elements.
            - If ')' is found, advance pos and return the populated lval.
            - If the string ends before ')', return an error.
5. **Parse Q-expressions**:
    - If the character is '{':
        - Create a new lval of type LVAL_QEXPR.
        - Advance pos past '{'.
        - Call lval_read_expr with end character '}'.
        - Process similarly to S-expressions, building a Q-expression lval.
6. **Parse Symbols and Numbers**:
    - If the character is a letter, digit, or special character (e.g., '+', '-', '*'):
        - Initialize an empty buffer to collect characters.
        - While the current character is alphanumeric or a valid symbol character (e.g., '+', '-', '/', etc.):
            - Append it to the buffer.
            - Advance pos.
        - Analyze the buffer:
            - If it’s a valid integer (e.g., "123" or "-45"), create an lval of type LVAL_NUM with that value.
            - Otherwise, create an lval of type LVAL_SYM with the buffer as the symbol name (e.g., "add").
7. **Parse Strings**:
    - If the character is '"':
        - Advance pos past '"'.
        - Initialize an empty buffer.
        - While the current character isn’t '"':
            - If it’s '', handle escape sequences:
                - '\n' → newline, '"' → quote, etc.

            - Advance pos twice (past '' and the next character).
        - Otherwise, append the character to the buffer and advance pos.
        - If the string ends before '"', return an error.
    - Advance past the closing '"'.
    - Create an lval of type LVAL_STR with the buffer contents.
8. **Handle Invalid Characters**:
    - If the character doesn't match any expected type (e.g., '#'), create an lval of type LVAL_ERR with a message like "Unexpected character: #".
9. **Finalize**:
    - Skip any trailing whitespace or comments after the parsed expression.
    - Return the constructed lval.

---

2. Evaluation Algorithm (lval_eval)

The evaluation algorithm takes an lval and processes it according to Lisp rules, using an environment (lenv) to resolve symbols and apply functions.

Steps:

1. **Check lval Type**:
    - Get the type of the input lval (e.g., LVAL_SYM, LVAL_SEXPR, etc.).
2. **Evaluate Symbols**:
    - If type is LVAL_SYM:
        - Search the environment (lenv) for the symbol's name.
        - Start at the current lenv:
            - Scan its list of symbol-value pairs.
            - If the symbol matches, return a copy of the associated lval.
        - If not found and there's a parent environment, repeat the search there.
        - If not found in any scope, return an lval of type LVAL_ERR with "Unbound symbol: <name>".
3. **Evaluate S-expressions**:
    - If type is LVAL_SEXPR:
        - If the S-expression is empty (no children), return it unchanged.
        - For each child lval in the S-expression:
            - Recursively call lval_eval on the child.
            - If any child returns an LVAL_ERR, propagate that error and stop.
            - Replace the child with its evaluated result.
        - After evaluation:

        - If only one child remains, return that child's lval.
        - Otherwise, treat the first child as a function and the rest as arguments.
        - Call lval_call with the function, environment, and arguments.
4. **Self-evaluating Types**:
    - For LVAL_NUM, LVAL_STR, LVAL_FUN, or LVAL_QEXPR:
        - Return the lval as-is (no further evaluation needed).

---

3. Function Application Algorithm (lval_call)

This algorithm applies a function (built-in or lambda) to a list of arguments.

Steps:

1. **Verify Function Type**:
    - Ensure the first lval is of type LVAL_FUN. If not, return an error like "First element must be a function".
2. **Handle Built-in Functions**:
    - If the function's builtin field is non-null:
        - Pass the environment (lenv) and the argument list (lval) to the built-in function pointer.
        - Execute the built-in logic (e.g., addition, list manipulation).
        - Return the result lval.
3. **Handle Lambda Functions**:
    - If the function's builtin field is null (lambda):
        - Extract the lambda's components: formals (parameter list), body (code to execute), and env (parent environment).
        - Compare the number of arguments to formals:
            - If too few, return a partial application (new lambda with remaining parameters).
            - If too many, return an error like "Too many arguments".
        - Handle variadic parameters:
            - If formals contains '&' followed by a symbol:
                - Bind all remaining arguments to that symbol as a Q-expression.
            - Otherwise, match arguments to formals one-to-one.
        - Create a new environment:
            - Copy the lambda's env as the parent.
            - For each formal parameter and argument pair:
                - Add the binding to the new environment using lenv_put.

        - Evaluate the body in the new environment:
            - Call lval_eval on the body lval.
            - Return the result.
4. **Error Checking**:
    - Validate formals syntax (e.g., '&' must be followed by exactly one symbol).
    - Return errors for mismatches or invalid formats.

---

4. Memory Management Algorithm (lval_del)

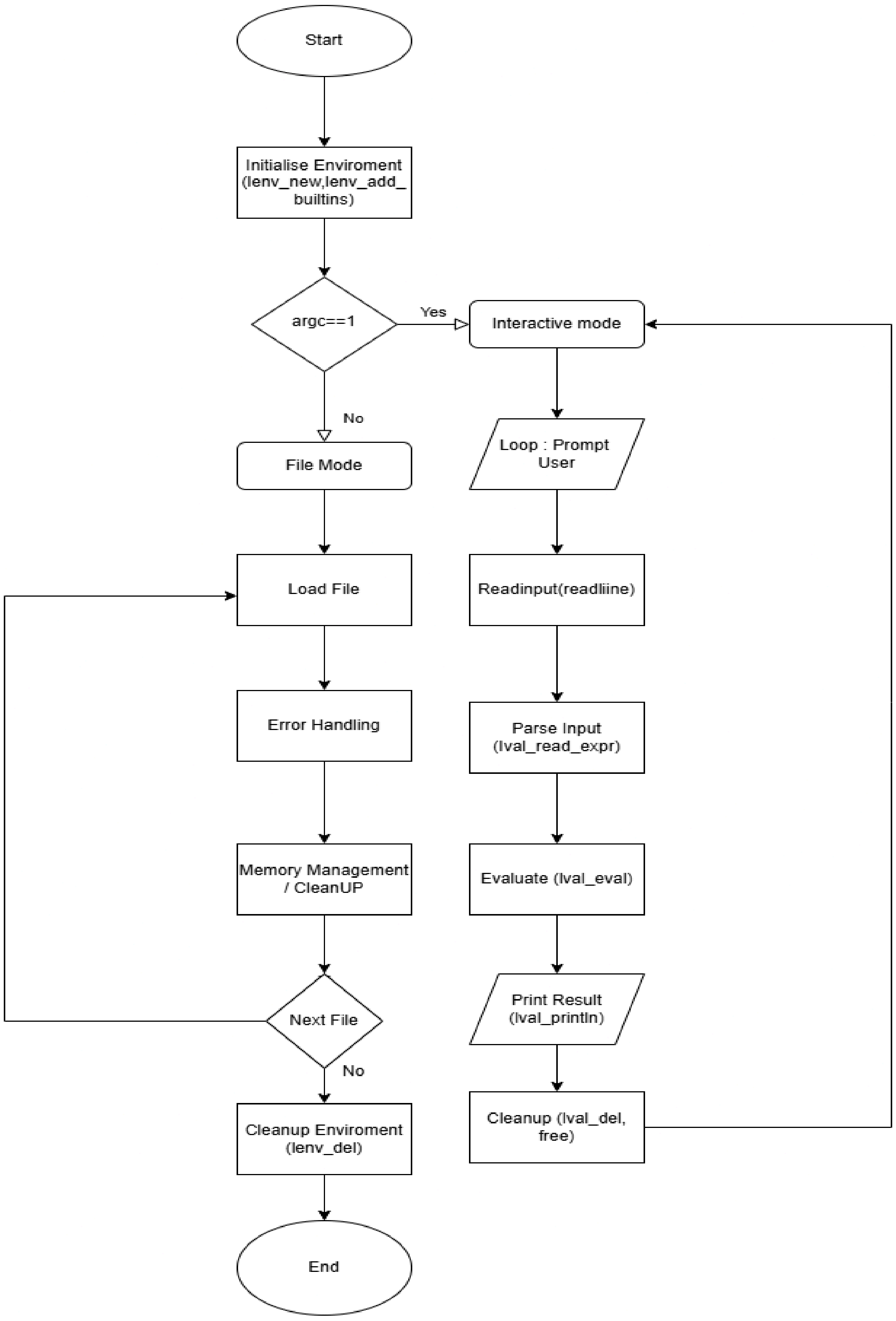
This algorithm ensures memory is freed properly to prevent leaks.

Steps:

1. **Check lval Type**:
    - Use a switch statement on the lval's type.
2. **Free Basic Types**:
    - LVAL_NUM: No additional memory; free the lval structure.
    - LVAL_ERR: Free the error message string, then the lval.
    - LVAL_SYM: Free the symbol string, then the lval.
    - LVAL_STR: Free the string data, then the lval.
3. **Free Functions**:
    - If builtin is null (lambda):
        - Recursively call lval_del on formals and body.
        - Free the lambda's environment (lenv).
    - Free the lval.
4. **Free Expressions**:
    - For LVAL_SEXPR or LVAL_QEXPR:
        - Iterate over the list of child lvals (stored in an array).
        - Call lval_del on each child.
        - Free the array of pointers.
        - Free the lval.
5. **Finalize**:
    - Free the lval structure itself using free().

.

4.2 Flowchart


Start
Initialise Enviroment (lenv_new,lenv_add_ builtins)
argc==1
Yes
Interactive mode
No
File Mode
Loop : Prompt User
Load File
Readinput(readliine)
Error Handling
Parse Input (lval_read_expr)
Memory Management / CleanUP
Evaluate (lval_eval)
Print Result (lval_println)
Next File
No
Cleanup Enviroment (lenv_del)
Cleanup (lval_del, free)
End

5. Implementation and Coding

5.1 Implementation

The interpreter centers on two key structures:

- **lval (Lisp Value)**: Represents data types (numbers, errors, symbols, strings, functions, S/Q-expressions) with fields for values, function pointers, and expression lists.
- **lenv (Lisp Environment)**: Manages symbol-value bindings with dynamic arrays and parent environment pointers for scoping.

The design is modular, with functions for constructing (lval_num, lval_err), manipulating (lval_add, lval_pop), and printing (lval_print) lval objects.

5.2 Coding for the Project

The code is organized as follows:

- **Platform-Specific Code**: Uses #ifdef _WIN32 for a basic readline on Windows; Unix systems use editline.
- **Data Structures**: Defines lval and lenv with enums and structs.
- **Construction**: Functions like lval_num, lval_sym, and lval_lambda create lval instances.
- **Memory Management**: lval_del and lval_copy handle deallocation and duplication.
- **List Operations**: lval_add, lval_join, lval_pop, and lval_take manipulate expression lists.
- **Printing**: lval_print and lval_print_str format output with escape handling.
- **Evaluation**: lval_eval and lval_eval_sexpr process expressions; lval_call executes functions.
- **Built-ins**: Functions like builtin_add, builtin_if, and builtin_load are registered via lenv_add_builtins.
- **Main**: Offers an interactive loop or file execution, initializing lenv and processing input.

Macros (LASSERT, LASSERT_TYPE) enhance error checking.

---

6. Result and Discussion

The interpreter correctly evaluates basic Lisp expressions (e.g., "(+ 1 2)" → 3) and supports built-in functions. The interactive prompt functions on both platforms, and file loading works for valid Lisp files. However, the Windows version lacks history support, and performance may falter with complex inputs due to reallocation overhead. Error reporting is effective, but unhandled edge cases could lead to memory leaks. The project lays a strong groundwork, with potential for further development.

---

7. Conclusion

"Backpain Version 0.0.0.1.1" achieves its goal of providing a functional Lisp interpreter in C, suitable for educational use or as a starting point for enhancements. Its cross-platform support and core feature set are strengths, though limitations like missing advanced features and basic Windows input handling suggest areas for improvement. Future iterations could add macros, optimize performance, and refine error handling, leveraging C’s portability and control.

---

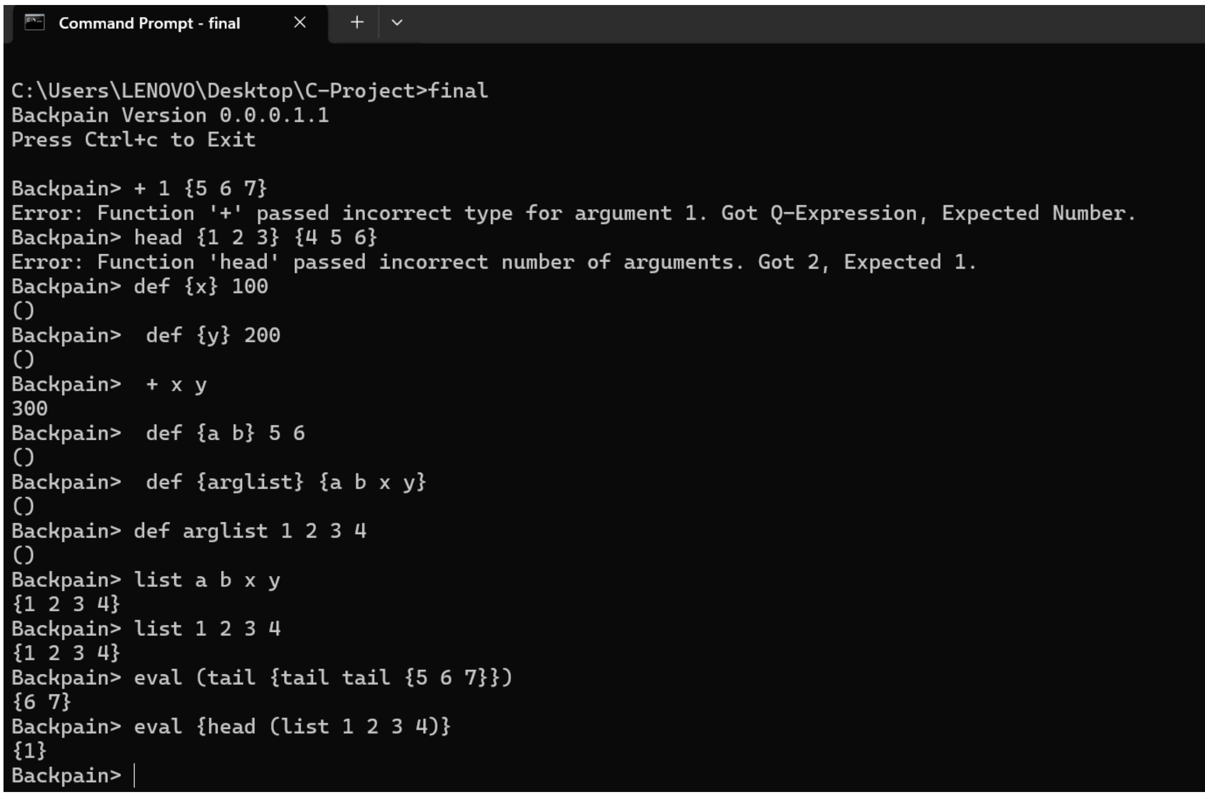
8. References

- Buildyourownlisp.com
- https://devdocs.io/c/\
- https://stackoverflow.com/questions/14155327/better-way-for-parser-combinators-in-c

---

9. Appendix

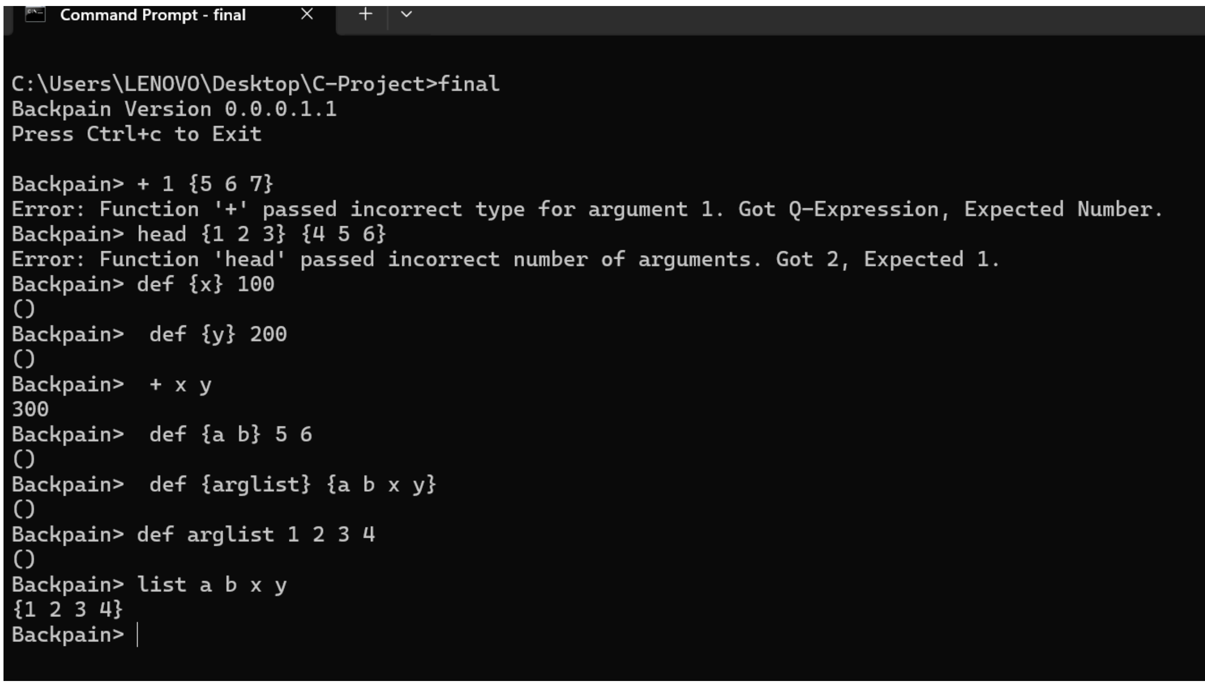
- **Sample Inputs and Outputs**:


Command Prompt - final

```
C:\Users\LENOVO\Desktop\C-Project>final
Backpain Version 0.0.0.1.1
Press Ctrl+c to Exit

Backpain> + 1 {5 6 7}
Error: Function '+' passed incorrect type for argument 1. Got Q-Expression, Expected Number.
Backpain> head {1 2 3} {4 5 6}
Error: Function 'head' passed incorrect number of arguments. Got 2, Expected 1.
Backpain> def {x} 100
()
Backpain>  def {y} 200
()
Backpain>  + x y
300
Backpain>  def {a b} 5 6
()
Backpain>  def {arglist} {a b x y}
()
Backpain> def arglist 1 2 3 4
()
Backpain> list a b x y
{1 2 3 4}
Backpain> list 1 2 3 4
{1 2 3 4}
Backpain> eval (tail {tail tail {5 6 7}})
{6 7}
Backpain> eval {head (list 1 2 3 4)}
{1}
Backpain>
```

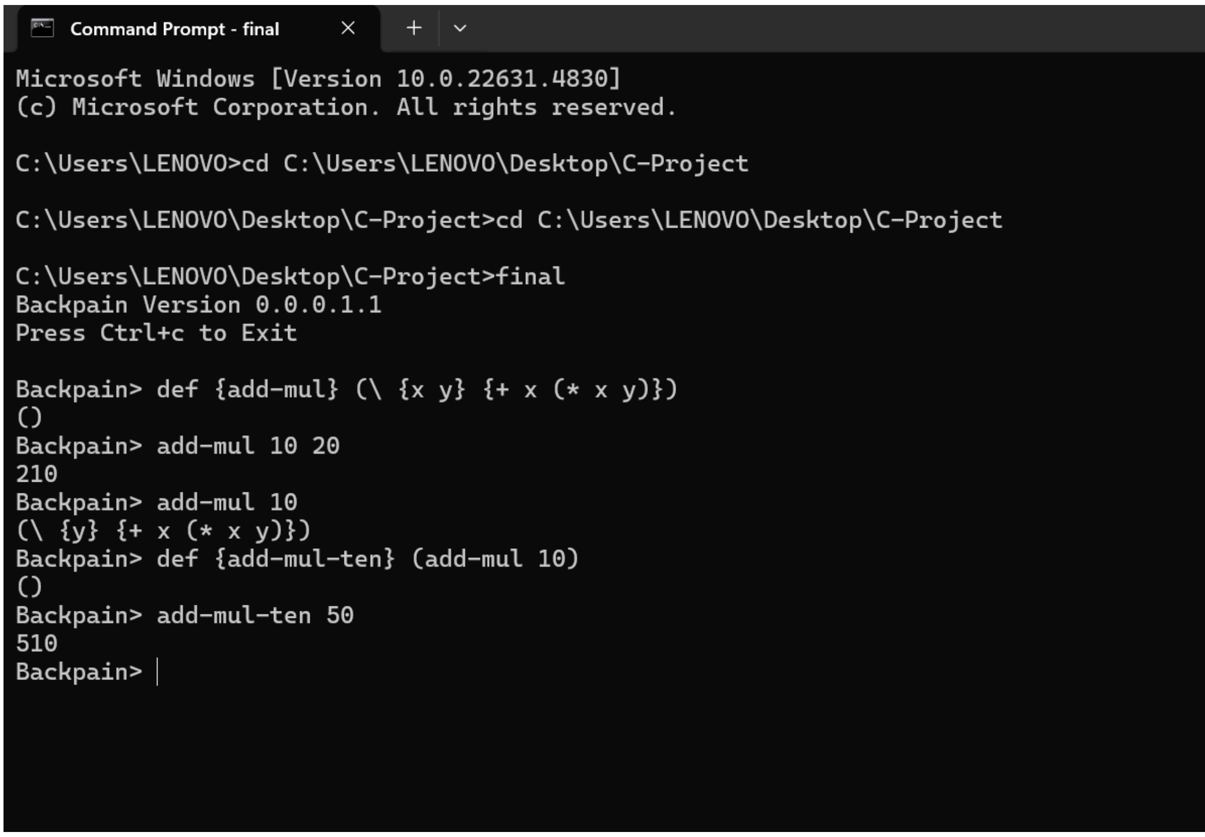
○



Command Prompt - final

```
C:\Users\LENOVO\Desktop\C-Project>final
Backpain Version 0.0.0.1.1
Press Ctrl+c to Exit

Backpain> + 1 {5 6 7}
Error: Function '+' passed incorrect type for argument 1. Got Q-Expression, Expected Number.
Backpain> head {1 2 3} {4 5 6}
Error: Function 'head' passed incorrect number of arguments. Got 2, Expected 1.
Backpain> def {x} 100
()
Backpain>  def {y} 200
()
Backpain>  + x y
300
Backpain>  def {a b} 5 6
()
Backpain>  def {arglist} {a b x y}
()
Backpain> def arglist 1 2 3 4
()
Backpain> list a b x y
{1 2 3 4}
Backpain>
```



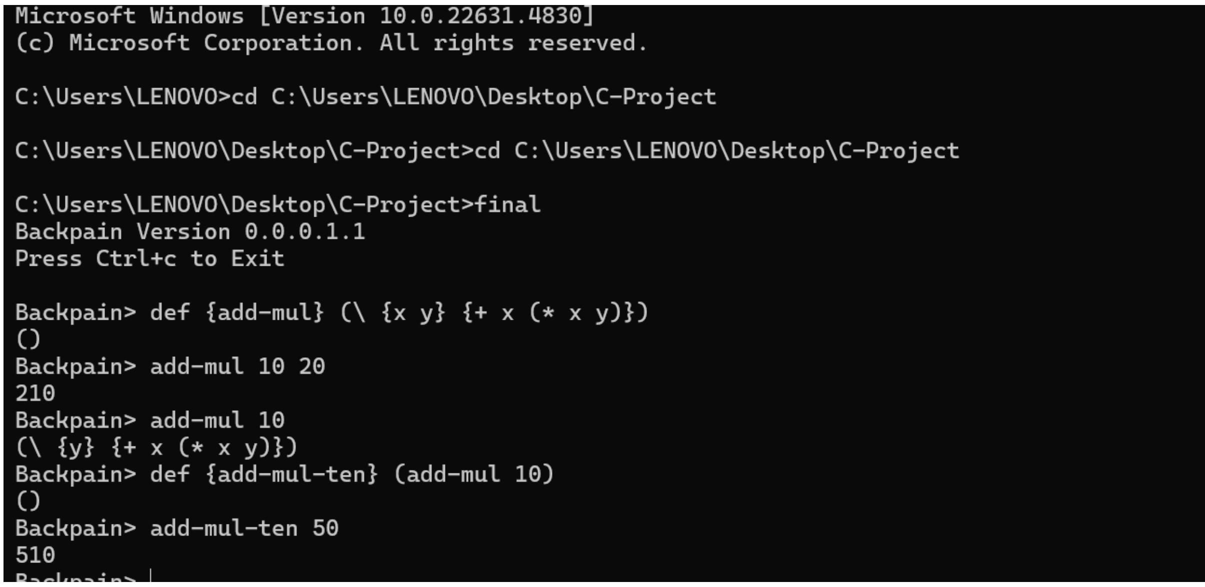
```
Microsoft Windows [Version 10.0.22631.4830]
(c) Microsoft Corporation. All rights reserved.

C:\Users\LENOVO>cd C:\Users\LENOVO\Desktop\C-Project

C:\Users\LENOVO\Desktop\C-Project>cd C:\Users\LENOVO\Desktop\C-Project

C:\Users\LENOVO\Desktop\C-Project>final
Backpain Version 0.0.0.1.1
Press Ctrl+c to Exit

Backpain> def {add-mul} (\ {x y} {+ x (* x y)})
()
Backpain> add-mul 10 20
210
Backpain> add-mul 10
(\ {y} {+ x (* x y)})
Backpain> def {add-mul-ten} (add-mul 10)
()
Backpain> add-mul-ten 50
510
Backpain>
```



```
Microsoft Windows [Version 10.0.22631.4830]
(c) Microsoft Corporation. All rights reserved.

C:\Users\LENOVO>cd C:\Users\LENOVO\Desktop\C-Project

C:\Users\LENOVO\Desktop\C-Project>cd C:\Users\LENOVO\Desktop\C-Project

C:\Users\LENOVO\Desktop\C-Project>final
Backpain Version 0.0.0.1.1
Press Ctrl+c to Exit

Backpain> def {add-mul} (\ {x y} {+ x (* x y)})
()
Backpain> add-mul 10 20
210
Backpain> add-mul 10
(\ {y} {+ x (* x y)})
Backpain> def {add-mul-ten} (add-mul 10)
()
Backpain> add-mul-ten 50
510
```

- **Source Code**: Full listing as provided.

```
#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdarg.h>
```

```
#include <errno.h>

#ifdef _WIN32

static char buffer[2048];

char* readline(char* prompt) {
  fputs(prompt, stdout);
  fgets(buffer, 2048, stdin);
  char* cpy = malloc(strlen(buffer)+1);
  strcpy(cpy, buffer);
  cpy[strlen(cpy)-1] = '\0';
  return cpy;
}

void add_history(char* unused) {}

#else
#include <editline/readline.h>
#include <editline/history.h>
#endif

/* Forward Declarations */

struct lval;
struct lenv;

typedef struct lval lval;

typedef struct lenv lenv;




/* Lisp Value */




enum { LVAL_ERR, LVAL_NUM,   LVAL_SYM, LVAL_STR,

       LVAL_FUN, LVAL_SEXPR, LVAL_QEXPR };


typedef lval*(*lbuiltin)(lenv*, lval*);


struct lval {

  int type;


  /* Basic */

  long num;

  char* err;

  char* sym;

  char* str;


  /* Function */
  lbuiltin builtin;

  lenv* env;

  lval* formals;

  lval* body;


  /* Expression */

  int count;

  lval** cell;

};


/* Construct a pointer to a new Number lval */

lval* lval_num(long x) {

  lval* v = malloc(sizeof(lval));

  v->type = LVAL_NUM;

  v->num = x;

  return v;

}


/* Construct a pointer to a Error lval */

lval* lval_err(char* fmt, ...) {

  lval* v = malloc(sizeof(lval));

  v->type = LVAL_ERR;

  va_list va;

  va_start(va, fmt);

  v->err = malloc(512);
  vsnprintf(v->err, 511, fmt, va);

  v->err = realloc(v->err, strlen(v->err)+1);

  va_end(va);

  return v;

}


/* Construct a pointer to a new Symbol lval */

lval* lval_sym(char* s) {

  lval* v = malloc(sizeof(lval));

  v->type = LVAL_SYM;

  v->sym = malloc(strlen(s) + 1);

  strcpy(v->sym, s);

  return v;

}


/* Construct a pointer to a new String lval */

lval* lval_str(char* s) {

  lval* v = malloc(sizeof(lval));

  v->type = LVAL_STR;

  v->str = malloc(strlen(s) + 1);

  strcpy(v->str, s);

  return v;

}


/* Construct a pointer to a new function lval */
```

```
lval* lval_builtin(lbuiltin func) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_FUN;
  v->builtin = func;
  return v;
}

lenv* lenv_new(void);

/* Construct a pointer to a new lambda function lval */
lval* lval_lambda(lval* formals, lval* body) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_FUN;
  v->builtin = NULL;
  v->env = lenv_new();
  v->formals = formals;
  v->body = body;
  return v;
}

/* A pointer to a new empty Sexpr lval */
lval* lval_sexpr(void) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_SEXPR;
  v->count = 0;
  v->cell = NULL;

  return v;

}


/* A pointer to a new empty qexpr lval */

lval* lval_qexpr(void) {

  lval* v = malloc(sizeof(lval));

  v->type = LVAL_QEXPR;

  v->count = 0;

  v->cell = NULL;

  return v;

}


/*preemptive declaration */

void lenv_del(lenv* e);


/*A function to delete lval */

void lval_del(lval* v) {


  switch (v->type) {

    case LVAL_NUM: break;

    case LVAL_FUN:

      if (!v->builtin) {

        lenv_del(v->env);

        lval_del(v->formals);
        lval_del(v->body);
      }
    break;
    case LVAL_ERR: free(v->err); break;
    case LVAL_SYM: free(v->sym); break;
    case LVAL_STR: free(v->str); break;
    case LVAL_QEXPR:
    case LVAL_SEXPR:
      for (int i = 0; i < v->count; i++) {
        lval_del(v->cell[i]);
      }
      free(v->cell);
    break;
  }

  free(v);
}

lenv* lenv_copy(lenv* e);

/*A function to copy lval */
lval* lval_copy(lval* v) {
  lval* x = malloc(sizeof(lval));
  x->type = v->type;
  switch (v->type) {
```

```
case LVAL_FUN:
  if (v->builtin) {
    x->builtin = v->builtin;
  } else {
    x->builtin = NULL;
    x->env = lenv_copy(v->env);
    x->formals = lval_copy(v->formals);
    x->body = lval_copy(v->body);
  }
break;
case LVAL_NUM: x->num = v->num; break;
case LVAL_ERR: x->err = malloc(strlen(v->err) + 1);
  strcpy(x->err, v->err);
break;
case LVAL_SYM: x->sym = malloc(strlen(v->sym) + 1);
  strcpy(x->sym, v->sym);
break;
case LVAL_STR: x->str = malloc(strlen(v->str) + 1);
  strcpy(x->str, v->str);
break;
case LVAL_SEXPR:
case LVAL_QEXPR:
  x->count = v->count;
  x->cell = malloc(sizeof(lval*) * x->count);
  for (int i = 0; i < x->count; i++) {
        x->cell[i] = lval_copy(v->cell[i]);
      }
    break;
  }
  return x;
}
```

/*A function to increase the count of the Exp list by 1 and use realloc to reallocate the space required by v->cell */

```
lval* lval_add(lval* v, lval* x) {
  v->count++;
  v->cell = realloc(v->cell, sizeof(lval*) * v->count);
  v->cell[v->count-1] = x;
  return v;
}
```

/* join Takes one or more Q-Expressions and returns a Q-Expression of them conjoined together */

```
lval* lval_join(lval* x, lval* y) {
  for (int i = 0; i < y->count; i++) {
    x = lval_add(x, y->cell[i]);
  }
  free(y->cell);
  free(y);
  return x;
}
```

```
/*Extracts the element at index i and shifts the rest of the list backward and returns
the extracted value */

lval* lval_pop(lval* v, int i) {

  lval* x = v->cell[i];

  memmove(&v->cell[i],

    &v->cell[i+1], sizeof(lval*) * (v->count-i-1));

  v->count--;

  v->cell = realloc(v->cell, sizeof(lval*) * v->count);

  return x;

}


/*The function removes an element from a list, deletes the list, and returns the
extracted element.*/

lval* lval_take(lval* v, int i) {

  lval* x = lval_pop(v, i);

  lval_del(v);

  return x;

}


void lval_print(lval* v);


/* function to print out S-Expressions types. i.e iterates through the elements in the
list, printing them with spaces between them (except the last element*/

void lval_print_expr(lval* v, char open, char close) {

  putchar(open);
  for (int i = 0; i < v->count; i++) {
    lval_print(v->cell[i]);
    if (i != (v->count-1)) {
      putchar(' ');
    }
  }
  putchar(close);
}

/* Possible unescapable characters */
char* lval_str_unescapable = "abfnrtv\\\'\"";

/* Function to unescape characters */
char lval_str_unescape(char x) {
  switch (x) {
    case 'a':  return '\a';
    case 'b':  return '\b';
    case 'f':  return '\f';
    case 'n':  return '\n';
    case 'r':  return '\r';
    case 't':  return '\t';
    case 'v':  return '\v';
    case '\\': return '\\';
    case '\'': return '\'';
    case '\"': return '\"';
  }
  return '\0';
}

/* List of possible escapable characters */
char* lval_str_escapable = "\a\b\f\n\r\t\v\\\'\"";

/* Function to escape characters */
char* lval_str_escape(char x) {
  switch (x) {
    case '\a': return "\\a";
    case '\b': return "\\b";
    case '\f': return "\\f";
    case '\n': return "\\n";
    case '\r': return "\\r";
    case '\t': return "\\t";
    case '\v': return "\\v";
    case '\\': return "\\\\";
    case '\'': return "\\\'";
    case '\"': return "\\\"";
  }
  return "";
}

/* function to print out String types.*/
```

```
void lval_print_str(lval* v) {
  putchar('"');
  /* Loop over the characters in the string */
  for (int i = 0; i < strlen(v->str); i++) {
    if (strchr(lval_str_escapable, v->str[i])) {
      /* If the character is escapable then escape it */
      printf("%s", lval_str_escape(v->str[i]));
    } else {
      /* Otherwise print character as it is */
      putchar(v->str[i]);
    }
  }
  putchar('"');
}

/* Main Iterated function for lvalue Printing .*/
void lval_print(lval* v) {
  switch (v->type) {
    case LVAL_FUN:
      if (v->builtin) {
        printf("<builtin>");
      } else {
        printf("(\\ ");
        lval_print(v->formals);
        putchar(' ');
        lval_print(v->body);
        putchar(')');
      }
    break;
    case LVAL_NUM:   printf("%li", v->num); break;
    case LVAL_ERR:   printf("Error: %s", v->err); break;
    case LVAL_SYM:   printf("%s", v->sym); break;
    case LVAL_STR:   lval_print_str(v); break;
    case LVAL_SEXPR: lval_print_expr(v, '(', ')'); break;
    case LVAL_QEXPR: lval_print_expr(v, '{', '}'); break;
  }
}

void lval_println(lval* v) { lval_print(v); putchar('\n'); }




/* Function to compare two Lisp values (lval objects) of any type .*/
int lval_eq(lval* x, lval* y) {

  if (x->type != y->type) { return 0; }

  switch (x->type) {
    case LVAL_NUM: return (x->num == y->num);
    case LVAL_ERR: return (strcmp(x->err, y->err) == 0);
    case LVAL_SYM: return (strcmp(x->sym, y->sym) == 0);
    case LVAL_STR: return (strcmp(x->str, y->str) == 0);
    case LVAL_FUN:
      if (x->builtin || y->builtin) {
        return x->builtin == y->builtin;
      } else {
        return lval_eq(x->formals, y->formals) && lval_eq(x->body, y->body);
      }
    case LVAL_QEXPR:
    case LVAL_SEXPR:
      if (x->count != y->count) { return 0; }
      for (int i = 0; i < x->count; i++) {
        if (!lval_eq(x->cell[i], y->cell[i])) { return 0; }
      }
      return 1;
    break;
  }
  return 0;
}

/* Function that returns the type of lisp (lval) objects */
char* ltype_name(int t) {
  switch(t) {
    case LVAL_FUN: return "Function";
    case LVAL_NUM: return "Number";
    case LVAL_ERR: return "Error";

    case LVAL_SYM: return "Symbol";

    case LVAL_STR: return "String";

    case LVAL_SEXPR: return "S-Expression";

    case LVAL_QEXPR: return "Q-Expression";

    default: return "Unknown";

  }

}




/* Lisp Environment */




struct lenv {

  lenv* par;

  int count;

  char** syms;

  lval** vals;

};


/*New  initialises the struct fields, while deletion iterates over the items in both lists
and deletes or frees them */

lenv* lenv_new(void) {
  lenv* e = malloc(sizeof(lenv));

  e->par = NULL;

  e->count = 0;

  e->syms = NULL;

  e->vals = NULL;

  return e;

}


void lenv_del(lenv* e) {

  for (int i = 0; i < e->count; i++) {

    free(e->syms[i]);

    lval_del(e->vals[i]);

  }

  free(e->syms);

  free(e->vals);

  free(e);

}
```

```
/* function that creates a new environment that is a duplicate of the original one, including all symbols and values. */

lenv* lenv_copy(lenv* e) {

  lenv* n = malloc(sizeof(lenv));

  n->par = e->par;

  n->count = e->count;
  n->syms = malloc(sizeof(char*) * n->count);
  n->vals = malloc(sizeof(lval*) * n->count);
  for (int i = 0; i < e->count; i++) {
    n->syms[i] = malloc(strlen(e->syms[i]) + 1);
    strcpy(n->syms[i], e->syms[i]);
    n->vals[i] = lval_copy(e->vals[i]);
  }
  return n;
}

/* functions that either get values from the environment or put values into it. */
lval* lenv_get(lenv* e, lval* k) {

  for (int i = 0; i < e->count; i++) {
    if (strcmp(e->syms[i], k->sym) == 0) { return lval_copy(e->vals[i]); }
  }

  if (e->par) {
    return lenv_get(e->par, k);
  } else {
    return lval_err("Unbound Symbol '%s'", k->sym);
  }
}

void lenv_put(lenv* e, lval* k, lval* v) {
  for (int i = 0; i < e->count; i++) {

    if (strcmp(e->syms[i], k->sym) == 0) {

      lval_del(e->vals[i]);

      e->vals[i] = lval_copy(v);

      return;

    }

  }


  e->count++;

  e->vals = realloc(e->vals, sizeof(lval*) * e->count);

  e->syms = realloc(e->syms, sizeof(char*) * e->count);

  e->vals[e->count-1] = lval_copy(v);

  e->syms[e->count-1] = malloc(strlen(k->sym)+1);

  strcpy(e->syms[e->count-1], k->sym);

}




/*define or update a symbol in the global scope */

void lenv_def(lenv* e, lval* k, lval* v) {

  while (e->par) { e = e->par; }

  lenv_put(e, k, v);

}
```

```
/* Builtins */



#define LASSERT(args, cond, fmt, ...) \
  if (!(cond)) { lval* err = lval_err(fmt, ##__VA_ARGS__); lval_del(args); return err; }

#define LASSERT_TYPE(func, args, index, expect) \
  LASSERT(args, args->cell[index]->type == expect, \
    "Function '%s' passed incorrect type for argument %i. Got %s, Expected %s.", \
    func, index, ltype_name(args->cell[index]->type), ltype_name(expect))

#define LASSERT_NUM(func, args, num) \
  LASSERT(args, args->count == num, \
    "Function '%s' passed incorrect number of arguments. Got %i, Expected %i.", \
    func, args->count, num)

#define LASSERT_NOT_EMPTY(func, args, index) \
  LASSERT(args, args->cell[index]->count != 0, \
    "Function '%s' passed {} for argument %i.", func, index);

lval* lval_eval(lenv* e, lval* v);
```

```
/*  defines a new lambda (anonymous function) in the environment */

lval* builtin_lambda(lenv* e, lval* a) {

  LASSERT_NUM("\\", a, 2);

  LASSERT_TYPE("\\", a, 0, LVAL_QEXPR);

  LASSERT_TYPE("\\", a, 1, LVAL_QEXPR);


  for (int i = 0; i < a->cell[0]->count; i++) {

    LASSERT(a, (a->cell[0]->cell[i]->type == LVAL_SYM),

      "Cannot define non-symbol. Got %s, Expected %s.",

      ltype_name(a->cell[0]->cell[i]->type), ltype_name(LVAL_SYM));

  }


  lval* formals = lval_pop(a, 0);

  lval* body = lval_pop(a, 0);

  lval_del(a);


  return lval_lambda(formals, body);

}


/*Creates a new list from the provided arguments.*/

lval* builtin_list(lenv* e, lval* a) {

  a->type = LVAL_QEXPR;

  return a;

}
```

```
/*Retrieves the first element of a list (or a Q-expression in this context). */

lval* builtin_head(lenv* e, lval* a) {

  LASSERT_NUM("head", a, 1);

  LASSERT_TYPE("head", a, 0, LVAL_QEXPR);

  LASSERT_NOT_EMPTY("head", a, 0);


  lval* v = lval_take(a, 0);

  while (v->count > 1) { lval_del(lval_pop(v, 1)); }

  return v;

}


/*Retrieves the last element of a list (or a Q-expression in this context). */

lval* builtin_tail(lenv* e, lval* a) {

  LASSERT_NUM("tail", a, 1);

  LASSERT_TYPE("tail", a, 0, LVAL_QEXPR);

  LASSERT_NOT_EMPTY("tail", a, 0);


  lval* v = lval_take(a, 0);

  lval_del(lval_pop(v, 0));

  return v;

}


/*It converts the Q-expression into an S-expression (evaluated list) and then
evaluates it using the lval_eval function. */

lval* builtin_eval(lenv* e, lval* a) {
  LASSERT_NUM("eval", a, 1);
  LASSERT_TYPE("eval", a, 0, LVAL_QEXPR);

  lval* x = lval_take(a, 0);
  x->type = LVAL_SEXPR;
  return lval_eval(e, x);
}

/*It ensures that all arguments are Q-expressions, then it pops and joins them one by
one.*/
lval* builtin_join(lenv* e, lval* a) {

  for (int i = 0; i < a->count; i++) {
    LASSERT_TYPE("join", a, i, LVAL_QEXPR);
  }

  lval* x = lval_pop(a, 0);

  while (a->count) {
    lval* y = lval_pop(a, 0);
    x = lval_join(x, y);
  }

  lval_del(a);
  return x;
```

```
}

/*Function that handles arithmetic operations */
lval* builtin_op(lenv* e, lval* a, char* op) {

  for (int i = 0; i < a->count; i++) {
    LASSERT_TYPE(op, a, i, LVAL_NUM);
  }

  lval* x = lval_pop(a, 0);

  if ((strcmp(op, "-") == 0) && a->count == 0) { x->num = -x->num; }

  while (a->count > 0) {
    lval* y = lval_pop(a, 0);

    if (strcmp(op, "+") == 0) { x->num += y->num; }
    if (strcmp(op, "-") == 0) { x->num -= y->num; }
    if (strcmp(op, "*") == 0) { x->num *= y->num; }
    if (strcmp(op, "/") == 0) {
      if (y->num == 0) {
        lval_del(x); lval_del(y);
        x = lval_err("Division By Zero.");
        break;
      }
      x->num /= y->num;
    }

    lval_del(y);
  }

  lval_del(a);
  return x;
}

lval* builtin_add(lenv* e, lval* a) { return builtin_op(e, a, "+"); }
lval* builtin_sub(lenv* e, lval* a) { return builtin_op(e, a, "-"); }
lval* builtin_mul(lenv* e, lval* a) { return builtin_op(e, a, "*"); }
lval* builtin_div(lenv* e, lval* a) { return builtin_op(e, a, "/"); }


/*Handles logic for checking symbols and defining and assigning variables*/
lval* builtin_var(lenv* e, lval* a, char* func) {
  LASSERT_TYPE(func, a, 0, LVAL_QEXPR);

  lval* syms = a->cell[0];
  for (int i = 0; i < syms->count; i++) {
    LASSERT(a, (syms->cell[i]->type == LVAL_SYM),
      "Function '%s' cannot define non-symbol. "
      "Got %s, Expected %s.",
      func, ltype_name(syms->cell[i]->type), ltype_name(LVAL_SYM));
  }

  LASSERT(a, (syms->count == a->count-1),
    "Function '%s' passed too many arguments for symbols. "
    "Got %i, Expected %i.",
    func, syms->count, a->count-1);

  for (int i = 0; i < syms->count; i++) {
    if (strcmp(func, "def") == 0) { lenv_def(e, syms->cell[i], a->cell[i+1]); }
    if (strcmp(func, "=")   == 0) { lenv_put(e, syms->cell[i], a->cell[i+1]); }
  }

  lval_del(a);
  return lval_sexpr();
}



/*  Used to define variables */
lval* builtin_def(lenv* e, lval* a) { return builtin_var(e, a, "def"); }

/* Used to assign value to variables */
lval* builtin_put(lenv* e, lval* a) { return builtin_var(e, a, "="); }
```

```
/*Comparision Operators */

lval* builtin_ord(lenv* e, lval* a, char* op) {

  LASSERT_NUM(op, a, 2);

  LASSERT_TYPE(op, a, 0, LVAL_NUM);

  LASSERT_TYPE(op, a, 1, LVAL_NUM);


  int r;

  if (strcmp(op, ">")  == 0) { r = (a->cell[0]->num >  a->cell[1]->num); }

  if (strcmp(op, "<")  == 0) { r = (a->cell[0]->num <  a->cell[1]->num); }

  if (strcmp(op, ">=") == 0) { r = (a->cell[0]->num >= a->cell[1]->num); }

  if (strcmp(op, "<=") == 0) { r = (a->cell[0]->num <= a->cell[1]->num); }

  lval_del(a);

  return lval_num(r);

}


lval* builtin_gt(lenv* e, lval* a) { return builtin_ord(e, a, ">");  }

lval* builtin_lt(lenv* e, lval* a) { return builtin_ord(e, a, "<");  }

lval* builtin_ge(lenv* e, lval* a) { return builtin_ord(e, a, ">="); }

lval* builtin_le(lenv* e, lval* a) { return builtin_ord(e, a, "<="); }



/*Comparision Operators */


lval* builtin_cmp(lenv* e, lval* a, char* op) {

  LASSERT_NUM(op, a, 2);
  int r;

  if (strcmp(op, "==") == 0) { r =  lval_eq(a->cell[0], a->cell[1]); }

  if (strcmp(op, "!=") == 0) { r = !lval_eq(a->cell[0], a->cell[1]); }

  lval_del(a);

  return lval_num(r);

}


lval* builtin_eq(lenv* e, lval* a) { return builtin_cmp(e, a, "=="); }

lval* builtin_ne(lenv* e, lval* a) { return builtin_cmp(e, a, "!="); }




/* If else statement */


lval* builtin_if(lenv* e, lval* a) {

  LASSERT_NUM("if", a, 3);

  LASSERT_TYPE("if", a, 0, LVAL_NUM);

  LASSERT_TYPE("if", a, 1, LVAL_QEXPR);

  LASSERT_TYPE("if", a, 2, LVAL_QEXPR);


  lval* x;

  a->cell[1]->type = LVAL_SEXPR;

  a->cell[2]->type = LVAL_SEXPR;


  if (a->cell[0]->num) {

    x = lval_eval(e, lval_pop(a, 1));
  } else {
    x = lval_eval(e, lval_pop(a, 2));
  }

  lval_del(a);
  return x;
}

/* Change forward declaration */
lval* lval_read_expr(char* s, int* i, char end);



/* Loads and executes a lisp file  */


lval* builtin_load(lenv* e, lval* a) {
  LASSERT_NUM("load", a, 1);
  LASSERT_TYPE("load", a, 0, LVAL_STR);

  /* Open file and check it exists */
  FILE* f = fopen(a->cell[0]->str, "rb");
  if (f == NULL) {
    lval* err = lval_err("Could not load Library %s", a->cell[0]->str);
    lval_del(a);
    return err;
  }
```

```
/* Read File Contents */
fseek(f, 0, SEEK_END);
long length = ftell(f);
fseek(f, 0, SEEK_SET);
char* input = calloc(length+1, 1);
fread(input, 1, length, f);
fclose(f);

/* Read from input to create an S-Expr */
int pos = 0;
lval* expr = lval_read_expr(input, &pos, '\0');
free(input);

/* Evaluate all expressions contained in S-Expr */
if (expr->type != LVAL_ERR) {
  while (expr->count) {
    lval* x = lval_eval(e, lval_pop(expr, 0));
    if (x->type == LVAL_ERR) { lval_println(x); }
    lval_del(x);
  }
} else {
  lval_println(expr);
}
```

```
  lval_del(expr);
  lval_del(a);

  return lval_sexpr();
}

/* Prints the output */

lval* builtin_print(lenv* e, lval* a) {
  for (int i = 0; i < a->count; i++) {
    lval_print(a->cell[i]); putchar(' ');
  }
  putchar('\n');
  lval_del(a);
  return lval_sexpr();
}

/* Returns an error type */

lval* builtin_error(lenv* e, lval* a) {
  LASSERT_NUM("error", a, 1);
  LASSERT_TYPE("error", a, 0, LVAL_STR);
  lval* err = lval_err(a->cell[0]->str);
  lval_del(a);
  return err;
```

```
}


/* registers a built-in function in the Lisp environment (lenv). It assigns a function
(func) to a symbol (name)  */

void lenv_add_builtin(lenv* e, char* name, lbuiltin func) {

  lval* k = lval_sym(name);

  lval* v = lval_builtin(func);

  lenv_put(e, k, v);

  lval_del(k); lval_del(v);

}




/*Registers all built-in functions into the Lisp environment */


void lenv_add_builtins(lenv* e) {

  /* Variable Functions */

  lenv_add_builtin(e, "\\",  builtin_lambda);

  lenv_add_builtin(e, "def", builtin_def);

  lenv_add_builtin(e, "=",   builtin_put);


  /* List Functions */

  lenv_add_builtin(e, "list", builtin_list);

  lenv_add_builtin(e, "head", builtin_head);

  lenv_add_builtin(e, "tail", builtin_tail);

  lenv_add_builtin(e, "eval", builtin_eval);
  lenv_add_builtin(e, "join", builtin_join);

  /* Mathematical Functions */
  lenv_add_builtin(e, "+", builtin_add);
  lenv_add_builtin(e, "-", builtin_sub);
  lenv_add_builtin(e, "*", builtin_mul);
  lenv_add_builtin(e, "/", builtin_div);

  /* Comparison Functions */
  lenv_add_builtin(e, "if", builtin_if);
  lenv_add_builtin(e, "==", builtin_eq);
  lenv_add_builtin(e, "!=", builtin_ne);
  lenv_add_builtin(e, ">",  builtin_gt);
  lenv_add_builtin(e, "<",  builtin_lt);
  lenv_add_builtin(e, ">=", builtin_ge);
  lenv_add_builtin(e, "<=", builtin_le);

  /* String Functions */
  lenv_add_builtin(e, "load",  builtin_load);
  lenv_add_builtin(e, "error", builtin_error);
  lenv_add_builtin(e, "print", builtin_print);
}

/* Evaluation */
```

```
/*Function responsible for calling (executing) a function in Lisp interpreter. */

lval* lval_call(lenv* e, lval* f, lval* a) {

  if (f->builtin) { return f->builtin(e, a); }

  int given = a->count;
  int total = f->formals->count;

  while (a->count) {

    if (f->formals->count == 0) {
      lval_del(a);
      return lval_err("Function passed too many arguments. "
        "Got %i, Expected %i.", given, total);
    }

    lval* sym = lval_pop(f->formals, 0);

    if (strcmp(sym->sym, "&") == 0) {

      if (f->formals->count != 1) {
        lval_del(a);
        return lval_err("Function format invalid. "
          "Symbol '&' not followed by single symbol.");
      }

      lval* nsym = lval_pop(f->formals, 0);
      lenv_put(f->env, nsym, builtin_list(e, a));
      lval_del(sym); lval_del(nsym);
      break;
    }

    lval* val = lval_pop(a, 0);
    lenv_put(f->env, sym, val);
    lval_del(sym); lval_del(val);
  }

  lval_del(a);

  if (f->formals->count > 0 &&
    strcmp(f->formals->cell[0]->sym, "&") == 0) {

    if (f->formals->count != 2) {
      return lval_err("Function format invalid. "
        "Symbol '&' not followed by single symbol.");
    }

    lval_del(lval_pop(f->formals, 0));
```

```
    lval* sym = lval_pop(f->formals, 0);

    lval* val = lval_qexpr();

    lenv_put(f->env, sym, val);

    lval_del(sym); lval_del(val);

  }


  if (f->formals->count == 0) {

    f->env->par = e;

    return builtin_eval(f->env, lval_add(lval_sexpr(), lval_copy(f->body)));

  } else {

    return lval_copy(f);

  }


}


/*Evaluates S-Expressions (functions and arguments) */

lval* lval_eval_sexpr(lenv* e, lval* v) {


  for (int i = 0; i < v->count; i++) { v->cell[i] = lval_eval(e, v->cell[i]); }

  for (int i = 0; i < v->count; i++) { if (v->cell[i]->type == LVAL_ERR) { return
lval_take(v, i); } }


  if (v->count == 0) { return v; }

  if (v->count == 1) { return lval_eval(e, lval_take(v, 0)); }
  lval* f = lval_pop(v, 0);
  if (f->type != LVAL_FUN) {
    lval* err = lval_err(
      "S-Expression starts with incorrect type. "
      "Got %s, Expected %s.",
      ltype_name(f->type), ltype_name(LVAL_FUN));
    lval_del(f); lval_del(v);
    return err;
  }

  lval* result = lval_call(e, f, v);
  lval_del(f);
  return result;
}

/*Evaluates any value (symbol lookup, number, S-Expression) */
lval* lval_eval(lenv* e, lval* v) {
  if (v->type == LVAL_SYM) {
    lval* x = lenv_get(e, v);
    lval_del(v);
    return x;
  }
  if (v->type == LVAL_SEXPR) { return lval_eval_sexpr(e, v); }
  return v;
}
```

```
/* Reading */


/*The function reads a symbol or number from the string s, checks if it is a valid
number or not, and then returns anlval (Lisp value) either as a symbol or a

 number. */

lval* lval_read_sym(char* s, int* i) {


  /* Allocate Empty String */

  char* part = calloc(1,1);


  /* While valid identifier characters */

  while (strchr(

      "abcdefghijklmnopqrstuvwxyz"

      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

      "0123456789_+-*\\/=<>!&", s[*i]) && s[*i] != '\0') {


    /* Append character to end of string */

    part = realloc(part, strlen(part)+2);

    part[strlen(part)+1] = '\0';

    part[strlen(part)+0] = s[*i];

    (*i)++;
  }

  /* Check if Identifier looks like number */
  int is_num = strchr("-0123456789", part[0]) != NULL;
  for (int j = 1; j < strlen(part); j++) {
    if (strchr("0123456789", part[j]) == NULL) { is_num = 0; break; }
  }
  if (strlen(part) == 1 && part[0] == '-') { is_num = 0; }

  /* Add Symbol or Number as lval */
  lval* x = NULL;
  if (is_num) {
    errno = 0;
    long v = strtol(part, NULL, 10);
    x = (errno != ERANGE) ? lval_num(v) : lval_err("Invalid Number %s", part);
  } else {
    x = lval_sym(part);
  }

  /* Free temp string */
  free(part);

  /* Return lval */
  return x;
}
```

/* function reads a string literal from an input, handling escape sequences, and returns it as an internal Lisp value (lval) of type string. */

```
lval* lval_read_str(char* s, int* i) {


  /* Allocate empty string */

  char* part = calloc(1,1);


  /* More forward one step past initial " character */

  (*i)++;

  while (s[*i] != '"') {


    char c = s[*i];


    /* If end of input then there is an unterminated string literal */

    if (c == '\0') {

      free(part);

      return lval_err("Unexpected end of input");

    }


    /* If backslash then unescape character after it */

    if (c == '\\') {

      (*i)++;

      /* Check next character is escapable */

      if (strchr(lval_str_unescapable, s[*i])) {
        c = lval_str_unescape(s[*i]);
      } else {
        free(part);
        return lval_err("Invalid escape sequence \\%c", s[*i]);
      }
    }

    /* Append character to string */
    part = realloc(part, strlen(part)+2);
    part[strlen(part)+1] = '\0';
    part[strlen(part)+0] = c;
    (*i)++;
  }
  /* Move forward past final " character */
  (*i)++;

  lval* x = lval_str(part);

  /* free temp string */
  free(part);

  return x;
}

lval* lval_read(char* s, int* i);
```

```
/* reads a sequence of Lisp values until it reaches a designated terminating
character, and then it returns an expression
(either a Q-expression or an S-expression) containing all those values. */
lval* lval_read_expr(char* s, int* i, char end) {

  /* Either create new qexpr or sexpr */
  lval* x = (end == '}') ? lval_qexpr() : lval_sexpr();

  /* While not at end character keep reading lvals */
  while (s[*i] != end) {
    lval* y = lval_read(s, i);
    /* If an error then return this and stop */
    if (y->type == LVAL_ERR) {
      lval_del(x);
      return y;
    } else {
      lval_add(x, y);
    }
  }

  /* Move past end character */
  (*i)++;

  return x;
```

```
}



/* Parses a single Lisp value from the input string, handling different data types
(S-expressions, Q-expressions, symbols, strings) and skipping

whitespace/comments. */

lval* lval_read(char* s, int* i) {


  /* Skip all trailing whitespace and comments */

  while (strchr(" \t\v\r\n;", s[*i]) && s[*i] != '\0') {

    if (s[*i] == ';') {

      while (s[*i] != '\n' && s[*i] != '\0') { (*i)++; }

    }

    (*i)++;

  }


  lval* x = NULL;


  /* If we reach end of input then we're missing something */

  if (s[*i] == '\0') {

    return lval_err("Unexpected end of input");

  }


  /* If next character is ( then read S-Expr */
  else if (s[*i] == '(') {
    (*i)++;
    x = lval_read_expr(s, i, ')');
  }

  /* If next character is { then read Q-Expr */
  else if (s[*i] == '{') {
    (*i)++;
    x = lval_read_expr(s, i, '}');
  }

  /* If next character is part of a symbol then read symbol */
  else if (strchr(
    "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "0123456789_+-*\\/=<>!&", s[*i])) {
    x = lval_read_sym(s, i);
  }

  /* If next character is " then read string */
  else if (strchr("\"", s[*i])) {
    x = lval_read_str(s, i);
  }

  /* Encountered some unexpected character */
```

```
  else {
    x = lval_err("Unexpected character %c", s[*i]);
  }

  /* Skip all trailing whitespace and comments */
  while (strchr(" \t\v\r\n;", s[*i]) && s[*i] != '\0') {
    if (s[*i] == ';') {
      while (s[*i] != '\n' && s[*i] != '\0') { (*i)++; }
    }
    (*i)++;
  }

  return x;

}

/* Main */

int main(int argc, char** argv) {

  lenv* e = lenv_new();
  lenv_add_builtins(e);

  /* Interactive Prompt */
  if (argc == 1) {
```

```
  puts("Backpain Version 0.0.0.1.1");
  puts("Press Ctrl+c to Exit\n");

  while (1) {

    char* input = readline("Backpain> ");
    add_history(input);

    /* Read from input to create an S-Expr */
    int pos = 0;
    lval* expr = lval_read_expr(input, &pos, '\0');

    /* Evaluate and print input */
    lval* x = lval_eval(e, expr);
    lval_println(x);
    lval_del(x);

    free(input);
  }
}

/* Supplied with list of files */
if (argc >= 2) {
  for (int i = 1; i < argc; i++) {
    lval* args = lval_add(lval_sexpr(), lval_str(argv[i]));

    lval* x = builtin_load(e, args);

    if (x->type == LVAL_ERR) { lval_println(x); }

    lval_del(x);

   }

  }


  lenv_del(e);


  return 0;

}
```

- **Debugging Tips**:
  - Verify pointer initialization and match malloc with free.
  - Check For implicit function declarations
  - Refer to buildyourownlisp.com
  - Refer to standard c library

The End