MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Centrul de Excelenţă în Informatică şi Tehnologii Informaţionale

STAGIULUI DE PRACTICĂ

Student: **Goreanu Victor**

Grupa: **W-2042**

Specialitatea: **Administrarea Aplicaţiilor WEB**

Baza de practică: **„SPARK SOLUTION” S.R.L.**

Conducătorul stagiului de pratică
de la unitatea econimică

Andrei Guzun

Conducătorul stagiului de practică
de la Centrul de Excelență

Zavadschi Vitalie

Chișinău 2024

Cuprins

Descrierea unității economice.

"SPARK SOLUTION" S.R.L. este o companie dinamică și inovatoare, specializată în furnizarea de soluții IT complexe și personalizate. Cu o echipă dedicată și creativă, ne străduim să oferim servicii care să răspundă precis cerințelor variate ale clienților noștri. Serviciile noastre cheie includ:

1. **Crearea aplicațiilor WEB**:
    - Dezvoltare de aplicații web adaptative și interactive, folosind cele mai noi tehnologii și practici din industrie.
    - Proiecte personalizate pentru diverse sectoare de activitate, asigurând performanță și securitate maximă.
2. **Optimizare SEO (Search Engine Optimization):**
    - Ne ocupăm de analiza cuvintelor cheie, optimizarea conținutului și structurii site-ului, precum și monitorizarea constantă și ajustarea strategiilor SEO.
    - Implementăm strategii SEO pentru a spori vizibilitatea site-urilor web în motoarele de căutare.
3. **Marketing**:
    - Dezvoltarea și implementarea de campanii de marketing digital, inclusiv publicitate online, marketing pe rețelele sociale și e-mail marketing.
    - Analiză și raportare detaliată a performanței campaniilor, pentru a asigura rezultate optime și a maximiza rentabilitatea investiției pentru clienți.

"SPARK SOLUTION" S.R.L. se dedică să fie un partener de încredere pentru clienții săi, oferind soluții inovatoare și eficiente menite să îmbunătățească prezența online și să contribuie la succesul pe termen lung. Echipa noastră de profesioniști cu experiență își folosește expertiza și creativitatea pentru a transforma ideile în realitate și a livra excelență în fiecare proiect.

Conținutul activităților și sarcinilor de lucru.

Planul de dezvoltare a proiectului

1. **Planificarea Inițială (22.04.2024 - 23.04.2024)**
    - Definirea obiectivelor proiectului: Clarificarea scopului și a funcționalităților de bază ale interpretului Lisp.
    - Stabilirea etapelor de dezvoltare: Identificarea principalelor etape și a sarcinilor necesare pentru finalizarea proiectului.
2. **Design şi Structură (24.04.2024 - 30.04.2024)**
    - Designul arhitecturii sistemului: Definirea componentelor principale ale interpretului Lisp, inclusiv parser-ul, evaluatorul și gestionarea erorilor.
    - Crearea schemelor de date: Definirea structurii datelor pentru expresiile Lisp și rezultatele evaluării.
3. **Dezvoltarea Parser-ului (01.05.2024 - 10.05.2024)**
    - Implementarea parser-ului folosind mpc: Crearea unui parser pentru a analiza și interpreta sintaxa Lisp.
    - Testarea parser-ului: Asigurarea corectitudinii și robusteței parser-ului prin teste unitare.
4. **Dezvoltarea Evaluator-ului (11.05.2024 - 20.05.2024)**
    - Implementarea funcțiilor de evaluare: Crearea mecanismului de evaluare pentru expresiile Lisp, incluzând suport pentru funcții aritmetice și logice.
    - Integrarea funcțiilor native: Adăugarea suportului pentru funcții predefinite (ex. +, -, *, /).
5. **Extinderea Funcţionalităţilor (21.05.2024 - 30.05.2024)**
    - Implementarea variabilelor și funcțiilor definite de utilizator: Permițând utilizatorilor să definească și să utilizeze propriile funcții și variabile.
    - Adăugarea suportului pentru liste: Implementarea manipulării listelor în Lisp.
6. **Gestionarea Erorilor (31.05.2024 - 05.06.2024)**
    - Implementarea mecanismelor de gestionare a erorilor: Asigurarea gestionării corespunzătoare a erorilor de sintaxă și de evaluare.
    - Testarea și debugging-ul: Identificarea și remedierea bug-urilor pentru a îmbunătăți stabilitatea interpretului.
7. **Optimizare şi Testare Finală (06.06.2024 - 15.06.2024)**

- Optimizarea performanței: Îmbunătățirea eficienței interpretului prin optimizări de cod și structuri de date.
- Testarea completă a funcționalităților: Asigurarea că toate funcțiile și caracteristicile sunt testate și funcționează corect.

8. **Documentare și Prezentare (16.06.2024 - 20.06.2024)**
    - Crearea documentației proiectului: Redactarea documentației tehnice și a ghidului de utilizare.
    - Pregătirea prezentării pentru consiliul școlii: Elaborarea unei prezentări detaliate a proiectului, incluzând obiectivele, implementarea și rezultatele.

Tehnologiile și instrumentelor utilizate.

Dezvoltarea Interpretoruluit

1. **C:**
    - Limbajul de programare utilizat pentru dezvoltarea interpretului Lisp. Este un limbaj de nivel scăzut, care oferă control detaliat asupra resurselor hardware și al memoriei.
2. **MPC (Micro Parser Combinators):**
    - Bibliotecă utilizată pentru a construi parser-ul Lisp. MPC permite definirea combinatorilor de parser pentru a analiza și interpreta sintaxa Lisp.
3. **Make:**
    - Instrument de automatizare a compilării și construcției proiectului. Make simplifică gestionarea și construirea codului sursă C prin definiția regulilor de compilare.

Gestionarea Erorilor și Debugging

4. **GDB (GNU Debugger):**
    - Debugger pentru limbajele de programare C și C++. Utilizat pentru identificarea și remedierea bug-urilor în codul sursă al interpretului Lisp.

Testare și Optimizare

5. **Unit Testing Frameworks:**
    - Framework-uri de testare utilizate pentru a crea și executa teste unitare pentru diferite componente ale interpretului, asigurând corectitudinea și stabilitatea acestuia.

Controlul Versiunilor

6. **Git:**
    - Sistem de control al versiunilor utilizat pentru a gestiona și urmări modificările în codul sursă al proiectului. Git asigură versionarea și colaborarea eficientă în echipă.

Editor de Cod și Mediul de Dezvoltare

7. **Neovim (nvim):**
    - Editor de cod avansat și extensibil utilizat pentru scrierea și editarea codului C. Neovim oferă funcționalități avansate de editare și integrare cu diverse plugin-uri.
8. **i3 Window Manager:**
    - Manager de ferestre pentru mediul desktop utilizat pe Ubuntu Linux. i3 oferă un management eficient al ferestrelor, optimizând fluxul de lucru al dezvoltatorului.
9. **Ubuntu Linux:**
    - Sistem de operare utilizat pentru dezvoltarea proiectului. Ubuntu Linux oferă un mediu robust și stabil pentru dezvoltarea software-ului.

Aceste tehnologii și instrumente au fost esențiale pentru dezvoltarea și implementarea interpretului Lisp, asigurând o dezvoltare eficientă, o gestionare corespunzătoare a erorilor și o documentație completă a proiectului.

Planificarea Proiectului de Dezvoltare a Interpretorului Lisp.

1. **Definirea Obiectivelor Proiectului**
    - **Scopul principal:** Crearea unui interpret Lisp funcțional, capabil să parcurgă, să analizeze și să execute cod Lisp.
    - **Obiective secundare:**
        - Implementarea unui parser robust pentru sintaxa Lisp.
        - Realizarea unui evaluator capabil să proceseze expresii și funcții Lisp.
        - Adăugarea suportului pentru variabile și funcții definite de utilizator.
        - Implementarea gestionării erorilor pentru a asigura robustețea și stabilitatea interpretului.
2. **Identificarea Utilizatorilor Țintă**
    - **Studenți:** Pentru învățarea și explorarea conceptelor de interpretoare și limbaje de programare.

- **Profesori:** Pentru a demonstra principiile limbajului Lisp și ale interpretoarelor în cadrul cursurilor.
- **Dezvoltatori:** Pentru utilizarea interpretului Lisp în proiecte de programare și cercetare.

3. **Stabilirea Funcționalităților Principale**
   - **Componente principale:** Parser, Evaluator, Sistem de gestionare a erorilor.
   - **Funcționalități:**
     - Analizarea și interpretarea sintaxei Lisp.
     - Evaluarea expresiilor aritmetice și logice.
     - Suport pentru funcții native și definite de utilizator.
     - Gestionarea variabilelor și listelor.
     - Sistem de gestionare a erorilor pentru a asigura robustetea interpretului.
4. **Crearea Wireframe-urilor și Mockup-urilor**
   - **Wireframe-uri:** Schițarea structurii logice a interpretului și a fluxului de date.
   - **Mockup-uri:** Design-ul vizual și funcțional al componentelor principale ale interpretului, incluzând structura parser-ului și a evaluator-ului.
5. **Planificarea Dezvoltării Interpretorului**
   - **Tehnologii utilizate:** C, MPC, Make.
   - **Etapele dezvoltării:**
     - Implementarea parser-ului cu ajutorul bibliotecii MPC.
     - Dezvoltarea evaluator-ului pentru procesarea și evaluarea expresiilor Lisp.
     - Integrarea funcțiilor native și definite de utilizator.
     - Implementarea gestionării variabilelor și listelor.
6. **Gestionarea Erorilor și Debugging**
   - **Tehnologii utilizate:** GDB.
   - **Etapele dezvoltării:**
     - Implementarea mecanismelor de gestionare a erorilor.
     - Debugging-ul codului pentru identificarea și remedierea bug-urilor.
     - Testarea și asigurarea corectitudinii funcționalităților implementate.
7. **Documentarea și Prezentarea**
   - **Tehnologii utilizate:** Doxygen, Markdown, Microsoft PowerPoint.
   - **Etapele dezvoltării:**
     - Crearea documentației tehnice a codului sursă.
     - Redactarea ghidului de utilizare a interpretului.
     - Pregătirea prezentării pentru consiliul școlii, incluzând obiectivele, implementarea și rezultatele proiectului.
8. **Controlul Versiunilor și Colaborarea**
   - **Tehnologii utilizate:** Git.
   - **Etapele dezvoltării:**
     - Configurarea repository-ului Git pentru gestionarea versiunilor codului.
     - Urmărirea modificărilor și colaborarea eficientă în echipă.
     - Asigurarea backup-ului și a restaurării codului în caz de necesitate.

9. **Editor de Cod şi Mediul de Dezvoltare**
    - **Tehnologii utilizate:** Neovim (nvim), i3 Window Manager, Ubuntu Linux.
    - **Etapele dezvoltării:**
        - Configurarea şi utilizarea Neovim pentru editarea şi dezvoltarea codului.
        - Utilizarea i3 Window Manager pentru un management eficient al ferestrelor.
        - Dezvoltarea pe Ubuntu Linux pentru un mediu de lucru robust şi stabil.
10. **Testarea şi Optimizarea**
    - **Tehnologii utilizate:** Unit Testing Frameworks.
    - **Etapele dezvoltării:**
        - Crearea şi executarea testelor unitare pentru verificarea funcţionalităţilor.
        - Optimizarea codului pentru îmbunătăţirea performanţei interpretului.
        - Asigurarea stabilităţii şi robusteţei prin teste extensive.
11. **Publicarea şi Mentenanţa**
    - **Etapele publicării:**
        - Publicarea codului pe un repository public pentru accesibilitate şi colaborare.
        - Monitorizarea performanţei interpretului şi rezolvarea rapidă a oricăror probleme.
        - Implementarea actualizărilor necesare şi îmbunătăţirea constantă a funcţionalităţilor interpretului.

Această planificare detaliată arată clar etapele şi tehnologiile utilizate pentru dezvoltarea interpretului Lisp, asigurând o prezentare comprehensivă şi structuratată pentru consiliul şcolii.

Codul sursă a aplicaţiilor elaborate.

Lisp.c

```c
#include "mpc.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

/* Parser Declariations */

mpc_parser_t* Number;
mpc_parser_t* Symbol;
mpc_parser_t* String;
mpc_parser_t* Comment;
mpc_parser_t* Sexpr;
mpc_parser_t* Qexpr;
mpc_parser_t* Expr;
mpc_parser_t* Lispy;

/* 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;
};

lval* lval_num(long x) {
 lval* v = malloc(sizeof(lval));
 v->type = LVAL_NUM;
 v->num = x;
 return v;
}

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;
}
```

```
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;
}

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;
}

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

lenv* lenv_new(void);

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;
}

lval* lval_sexpr(void) {
 lval* v = malloc(sizeof(lval));
 v->type = LVAL_SEXPR;
```

```
 v->count = 0;
 v->cell = NULL;
 return v;
}

lval* lval_qexpr(void) {
 lval* v = malloc(sizeof(lval));
 v->type = LVAL_QEXPR;
 v->count = 0;
 v->cell = NULL;
 return v;
}

void lenv_del(lenv* e);

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);

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;
```

```
}

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;
}

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;
}

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;
}

lval* lval_take(lval* v, int i) {
 lval* x = lval_pop(v, i);
 lval_del(v);
 return x;
}

void lval_print(lval* v);

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);
}

void lval_print_str(lval* v) {
  /* Make a Copy of the string */
  char* escaped = malloc(strlen(v->str)+1);
  strcpy(escaped, v->str);
  /* Pass it through the escape function */
  escaped = mpcf_escape(escaped);
  /* Print it between " characters */
  printf("\"%s\"", escaped);
  /* free the copied string */
  free(escaped);
}

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'); }

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;
}

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;
};

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);
}

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;
}

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);
}

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);

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);
}

lval* builtin_list(lenv* e, lval* a) {
 a->type = LVAL_QEXPR;
 return a;
}

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;
}

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;
}

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);
}

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;
}

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, "/"); }

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();
}

lval* builtin_def(lenv* e, lval* a) { return builtin_var(e, a, "def"); }
lval* builtin_put(lenv* e, lval* a) { return builtin_var(e, a, "="); }

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, "<="); }

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, "!="); }

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;
}

lval* lval_read(mpc_ast_t* t);

lval* builtin_load(lenv* e, lval* a) {
 LASSERT_NUM("load", a, 1);
 LASSERT_TYPE("load", a, 0, LVAL_STR);
  /* Parse File given by string name */
 mpc_result_t r;
 if (mpc_parse_contents(a->cell[0]->str, Lispy, &r)) {

   /* Read contents */
   lval* expr = lval_read(r.output);
   mpc_ast_delete(r.output);

   /* Evaluate each Expression */
   while (expr->count) {
     lval* x = lval_eval(e, lval_pop(expr, 0));
     /* If Evaluation leads to error print it */
     if (x->type == LVAL_ERR) { lval_println(x); }
     lval_del(x);
   }

   /* Delete expressions and arguments */
   lval_del(expr);
   lval_del(a);

   /* Return empty list */
   return lval_sexpr();

 } else {
   /* Get Parse Error as String */
   char* err_msg = mpc_err_string(r.error);
   mpc_err_delete(r.error);

   /* Create new error message using it */
```

```
    lval* err = lval_err("Could not load Library %s", err_msg);
    free(err_msg);
    lval_del(a);

    /* Cleanup and return error */
    return err;
  }
}

lval* builtin_print(lenv* e, lval* a) {
  /* Print each argument followed by a space */
  for (int i = 0; i < a->count; i++) {
    lval_print(a->cell[i]); putchar(' ');
  }
  /* Print a newline and delete arguments */
  putchar('\n');
  lval_del(a);
  return lval_sexpr();
}

lval* builtin_error(lenv* e, lval* a) {
  LASSERT_NUM("error", a, 1);
  LASSERT_TYPE("error", a, 0, LVAL_STR);
  /* Construct Error from first argument */
  lval* err = lval_err(a->cell[0]->str);
  /* Delete arguments and return */
  lval_del(a);
  return err;
}

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);
}

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 */

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);
  }
  }

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;
}

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 */

lval* lval_read_num(mpc_ast_t* t) {
 errno = 0;
 long x = strtol(t->contents, NULL, 10);
 return errno != ERANGE ? lval_num(x) : lval_err("Invalid Number.");
}

lval* lval_read_str(mpc_ast_t* t) {
 /* Cut off the final quote character */
 t->contents[strlen(t->contents)-1] = '\0';
 /* Copy the string missing out the first quote character */
 char* unescaped = malloc(strlen(t->contents+1)+1);
 strcpy(unescaped, t->contents+1);
 /* Pass through the unescape function */
 unescaped = mpcf_unescape(unescaped);
 /* Construct a new lval using the string */
 lval* str = lval_str(unescaped);
 /* Free the string and return */
 free(unescaped);
 return str;
}

lval* lval_read(mpc_ast_t* t) {
  if (strstr(t->tag, "number")) { return lval_read_num(t); }
 if (strstr(t->tag, "string")) { return lval_read_str(t); }
 if (strstr(t->tag, "symbol")) { return lval_sym(t->contents); }
  lval* x = NULL;
 if (strcmp(t->tag, ">") == 0) { x = lval_sexpr(); }
 if (strstr(t->tag, "sexpr"))  { x = lval_sexpr(); }
 if (strstr(t->tag, "qexpr"))  { x = lval_qexpr(); }
  for (int i = 0; i < t->children_num; i++) {
   if (strcmp(t->children[i]->contents, "(") == 0) { continue; }
   if (strcmp(t->children[i]->contents, ")") == 0) { continue; }
```

```
    if (strcmp(t->children[i]->contents, "}") == 0) { continue; }
    if (strcmp(t->children[i]->contents, "{") == 0) { continue; }
    if (strcmp(t->children[i]->tag,  "regex") == 0) { continue; }
    if (strstr(t->children[i]->tag, "comment")) { continue; }
    x = lval_add(x, lval_read(t->children[i]));
  }
  return x;
}

/* Main */

int main(int argc, char** argv) {
  Number  = mpc_new("number");
  Symbol  = mpc_new("symbol");
  String  = mpc_new("string");
  Comment = mpc_new("comment");
  Sexpr   = mpc_new("sexpr");
  Qexpr   = mpc_new("qexpr");
  Expr    = mpc_new("expr");
  Lispy   = mpc_new("lispy");
  mpca_lang(MPCA_LANG_DEFAULT,
    "                                          \
      number  : /-?[0-9]+/ ;                   \
      symbol  : /[a-zA-Z0-9_+\\-*\\/\\\\=<>!&]+/ ; \
      string  : /\"(\\\\.|[^\"])*\"/ ;         \
      comment : /;[^\\r\\n]*/ ;                \
      sexpr   : '(' <expr>* ')' ;              \
      qexpr   : '{' <expr>* '}' ;              \
      expr    : <number>  | <symbol> | <string>    \
              | <comment> | <sexpr>  | <qexpr>;    \
      lispy   : /^/ <expr>* /$/ ;              \
    ",
    Number, Symbol, String, Comment, Sexpr, Qexpr, Expr, Lispy);
  lenv* e = lenv_new();
  lenv_add_builtins(e);
  //  builtin_load(e, "library.lisp");

  /* Interactive Prompt */
```

```
if (argc == 1) {

  puts("Lispy Version 0.0.0.1.0");
  puts("Press Ctrl+c to Exit\n");
   while (1) {

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

    mpc_result_t r;
    if (mpc_parse("<stdin>", input, Lispy, &r)) {

      lval* x = lval_eval(e, lval_read(r.output));
      lval_println(x);
      lval_del(x);

      mpc_ast_delete(r.output);
    } else {
      mpc_err_print(r.error);
      mpc_err_delete(r.error);
    }

    free(input);

  }
}
 /* Supplied with list of files */
if (argc >= 2) {
   /* loop over each supplied filename (starting from 1) */
  for (int i = 1; i < argc; i++) {

    /* Argument list with a single argument, the filename */
    lval* args = lval_add(lval_sexpr(), lval_str(argv[i]));

    /* Pass to builtin load and get the result */
    lval* x = builtin_load(e, args);

    /* If the result is an error be sure to print it */
```

```
    if (x->type == LVAL_ERR) { lval_println(x); }
    lval_del(x);
  }
 }
  lenv_del(e);
  mpc_cleanup(8,
   Number, Symbol, String, Comment,
   Sexpr,  Qexpr,  Expr,   Lispy);
  return 0;
}
```

library.lisp

```
;;;
;;;   Lispy Standard Prelude
;;;

;;; Atoms
(def {nil} {})
(def {true} 1)
(def {false} 0)

;;; Functional Functions

; Function Definitions
(def {fun} (\ {f b} {
 def (head f) (\ (tail f) b)
}))

; Open new scope
(fun {let b} {
 ((\ {_} b) ())
})

; Unpack List to Function
(fun {unpack f l} {
```

```
 eval (join (list f) l)
})

; Unapply List to Function
(fun {pack f & xs} {f xs})

; Curried and Uncurried calling
(def {curry} unpack)
(def {uncurry} pack)

; Perform Several things in Sequence
(fun {do & l} {
 if (== l nil)
   {nil}
   {last l}
})

;;; Logical Functions

; Logical Functions
(fun {not x}   {- 1 x})
(fun {or x y}  {+ x y})
(fun {and x y} {* x y})


;;; Numeric Functions

; Minimum of Arguments
(fun {min & xs} {
 if (== (tail xs) nil) {fst xs}
   {do
     (= {rest} (unpack min (tail xs)))
     (= {item} (fst xs))
     (if (< item rest) {item} {rest})
   }
})

; Maximum of Arguments
```

```
(fun {max & xs} {
 if (== (tail xs) nil) {fst xs}
   {do
     (= {rest} (unpack max (tail xs)))
     (= {item} (fst xs))
     (if (> item rest) {item} {rest})
   }
})

;;; Conditional Functions

(fun {select & cs} {
 if (== cs nil)
   {error "No Selection Found"}
   {if (fst (fst cs)) {snd (fst cs)} {unpack select (tail cs)}}
})

(fun {case x & cs} {
 if (== cs nil)
   {error "No Case Found"}
   {if (== x (fst (fst cs))) {snd (fst cs)} {
     unpack case (join (list x) (tail cs))}}
})

(def {otherwise} true)



;;; Misc Functions

(fun {flip f a b} {f b a})
(fun {ghost & xs} {eval xs})
(fun {comp f g x} {f (g x)})

;;; List Functions

; First, Second, or Third Item in List
(fun {fst l} { eval (head l) })
(fun {snd l} { eval (head (tail l)) })
```

```
(fun {trd l} { eval (head (tail (tail l))) })

; List Length
(fun {len l} {
 if (== l nil)
   {0}
   {+ 1 (len (tail l))}
})

; Nth item in List
(fun {nth n l} {
 if (== n 0)
   {fst l}
   {nth (- n 1) (tail l)}
})

; Last item in List
(fun {last l} {nth (- (len l) 1) l})

; Apply Function to List
(fun {map f l} {
 if (== l nil)
   {nil}
   {join (list (f (fst l))) (map f (tail l))}
})

; Apply Filter to List
(fun {filter f l} {
 if (== l nil)
   {nil}
   {join (if (f (fst l)) {head l} {nil}) (filter f (tail l))}
})

; Return all of list but last element
(fun {init l} {
 if (== (tail l) nil)
   {nil}
   {join (head l) (init (tail l))}
```

```
})

; Reverse List
(fun {reverse l} {
 if (== l nil)
   {nil}
   {join (reverse (tail l)) (head l)}
})

; Fold Left
(fun {foldl f z l} {
 if (== l nil)
   {z}
   {foldl f (f z (fst l)) (tail l)}
})

; Fold Right
(fun {foldr f z l} {
 if (== l nil)
   {z}
   {f (fst l) (foldr f z (tail l))}
})

(fun {sum l} {foldl + 0 l})
(fun {product l} {foldl * 1 l})

; Take N items
(fun {take n l} {
 if (== n 0)
   {nil}
   {join (head l) (take (- n 1) (tail l))}
})

; Drop N items
(fun {drop n l} {
 if (== n 0)
   {l}
   {drop (- n 1) (tail l)}
```

```
})

; Split at N
(fun {split n l} {list (take n l) (drop n l)})

; Take While
(fun {take-while f l} {
 if (not (unpack f (head l)))
   {nil}
   {join (head l) (take-while f (tail l))}
})

; Drop While
(fun {drop-while f l} {
 if (not (unpack f (head l)))
   {l}
   {drop-while f (tail l)}
})

; Element of List
(fun {elem x l} {
 if (== l nil)
   {false}
   {if (== x (fst l)) {true} {elem x (tail l)}}
})

; Find element in list of pairs
(fun {lookup x l} {
 if (== l nil)
   {error "No Element Found"}
   {do
     (= {key} (fst (fst l)))
     (= {val} (snd (fst l)))
     (if (== key x) {val} {lookup x (tail l)})
   }
})

; Zip two lists together into a list of pairs
```

```
(fun {zip x y} {
 if (or (== x nil) (== y nil))
   {nil}
   {join (list (join (head x) (head y))) (zip (tail x) (tail y))}
})

; Unzip a list of pairs into two lists
(fun {unzip l} {
 if (== l nil)
   {{nil nil}}
   {do
     (= {x} (fst l))
     (= {xs} (unzip (tail l)))
     (list (join (head x) (fst xs)) (join (tail x) (snd xs)))
   }
})

;;; Other Fun

; Fibonacci
(fun {fib n} {
 select
   { (== n 0) 0 }
   { (== n 1) 1 }
   { otherwise (+ (fib (- n 1)) (fib (- n 2))) }
})
```

Testarea Interpretorului Lisp

1. **Metoda de Testare Utilizată**
    - **Testare Manuală:** Pentru a asigura corectitudinea și stabilitatea interpretorului Lisp, s-a efectuat o testare manuală detaliată. Această abordare a permis identificarea erorilor și verificarea funcționalităților într-un mod meticulos.
2. **Obiectivele Testării Manuale**
    - Verificarea corectitudinii funcționalităților principale ale interpretorului.

- Evaluarea experienței dezvoltatorului în utilizarea interpretorului și a funcționalităților acestuia.
- Asigurarea compatibilității cu diferitele sisteme de operare și configurări de mediu.

3. **Scenarii de Testare și Verificări**
   - **Analiza Sintaxei Lisp:**
     - Verificarea capacitatii de a parsa corect sintaxa Lisp.
     - Testarea evaluarii expresiilor Lisp.
   - **Gestionarea Variabilelor:**
     - Verificarea adaugarii, accesarii si stergerea variabilelor.
   - **Suport pentru Funcții Definite de Utilizator:**
     - Testarea definirii și utilizării funcțiilor de către utilizator.
   - **Gestionarea Erorilor:**
     - Verificarea comportamentului interpretorului în fața unor erori de sintaxă sau execuție.
   - **Compatibilitate cu Diferite Configurații de Mediu:**
     - Testarea interpretorului în diferite sisteme de operare și configurații de mediu pentru a asigura funcționarea corectă.
4. **Rezultatele Testării Manuale**
   - **Funcționalități Corecte:** Majoritatea funcționalităților interpretorului au fost validate cu succes, asigurând un comportament corect și consistent.
   - **Erori Minore:** Au fost identificate și remediate erori minore legate de parsarea și evaluarea expresiilor.
   - **Compatibilitate:** Interpretorul a funcționat corespunzător în diferite medii de testare, inclusiv în sisteme de operare variate și cu diferite configurații de mediu.

Această metodă riguroasă de testare manuală a contribuit la îmbunătățirea calității și fiabilității interpretorului Lisp, asigurând că acesta răspunde cerințelor și așteptărilor utilizatorilor.

Concluzii

În concluzie, proiectul de dezvoltare a interpretorului Lisp a reprezentat o experiență valoroasă și îmbucurătoare. Prin implicarea în acest proiect, am avut oportunitatea de a explora în profunzime concepte esențiale de limbaje de programare și de a dobândi abilități practice în dezvoltarea și testarea software-ului.

De la definirea obiectivelor inițiale până la finalizarea și testarea interpretorului, am fost ghidați de o metodologie riguroasă și de o atenție deosebită la detalii. Testarea manuală meticuloasă ne-a permis să identificăm și să corectăm erori, asigurând astfel o funcționalitate corectă și robustă a interpretorului.

Cu toate că am întâmpinat unele provocări pe parcursul proiectului, precum gestionarea corectă a erorilor și asigurarea compatibilității cu diferite medii de testare, am reușit să depășim obstacolele și să obținem rezultatele dorite.

În final, interpretorul Lisp reprezintă un produs de care suntem mândri și care demonstrează angajamentul nostru față de excelență în dezvoltarea software-ului. Acest proiect nu numai că ne-a oferit oportunitatea de a aplica cunoștințele teoretice acumulate, dar ne-a și consolidat încrederea în abilitățile noastre și ne-a pregătit pentru provocările viitoare în domeniul tehnologiei.

Bibliografie

Documentație Tehnică

- "Build Your Own Lisp Documentation"
  - Oferă informații detaliate despre structura, implementarea și utilizarea interpretorului Lisp.
- "Learn C The Hard Way" by Zed A. Shaw
  - Oferă un ghid detaliat pentru începători în programarea în limbajul C, necesară pentru dezvoltarea interpretorului.
- "MPC Parser Combinator Library Documentation"
  - Documentație pentru MPC, biblioteca folosită pentru parsarea sintaxei Lisp în cadrul interpretorului.

Resurse Online și Tutoriale

- "Learn Lisp Programming" by Lisp Journey
  - O serie de tutoriale și exemple practice pentru a învăța și aprofunda cunoștințele despre programarea în Lisp.
- "C Programming Tutorial" by Programiz
  - Un tutorial complet pentru a învăța și practica programarea în limbajul C, necesară pentru dezvoltarea interpretorului Lisp.

Instrumente și Tehnologii Utilizate

- C Programming Language
  - Limbajul de programare fundamental folosit pentru implementarea interpretorului Lisp.
- MPC Parser Combinator Library
  - Biblioteca folosită pentru a construi parser-ul necesar pentru interpretarea sintaxei Lisp.
- GNU Compiler Collection (GCC)

    - Compilatorul folosit pentru a compila și executa codul sursă scris în limbajul C.
- Linux Operating System
    - Sistemul de operare utilizat pentru dezvoltarea și testarea interpretorului, oferind un mediu de lucru stabil și eficient.

Anexe

```
lispy> + 1 (* 7 5) 3
39
lispy> (- 100)
-100
lispy>
()
lispy> /
/
lispy> (/ ())
Error: Cannot operate on non-number!
lispy>

lispy> list 1 2 3 4
{1 2 3 4}
lispy> {head (list 1 2 3 4)}
{head (list 1 2 3 4)}
lispy> eval {head (list 1 2 3 4)}
{1}
lispy> tail {tail tail tail}
{tail tail}
lispy> eval (tail {tail tail {5 6 7}})
{6 7}
lispy> eval (head {(+ 1 2) (+ 10 20)})
3
```

```
lispy> def {x} 100
()
lispy> def {y} 200
()
lispy> x
100
lispy> y
200
lispy> + x y
300
lispy> def {a b} 5 6
()
lispy> + a b
11
lispy> def {arglist} {a b x y}
()
lispy> arglist
{a b x y}
lispy> def arglist 1 2 3 4
()
lispy> list a b x y
{1 2 3 4}
```

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

```
lispy> > 10 5
1
lispy> <= 88 5
0
lispy> == 5 6
0
lispy> == 5 {}
0
lispy> == 1 1
1
lispy> != {} 56
1
lispy> == {1 2 3 {5 6}} {1   2  3   {5 6}}
1
lispy> def {x y} 100 200
()
lispy> if (== x y) {+ x y} {- x y}
-100
```

```
lispy> print "Hello World!"
"Hello World!"
()
lispy> error "This is an error"
Error: This is an error
lispy> load "hello.lspy"
"Hello World!"
()
```

```
lispy> load "library"
()
lispy> fun
(\ {f b} {def (head f) (\ (tail f) b)})
lispy> eval (head { "CETI!" "Goreanu Victor"})
"CETI!"
lispy> eval {+ 10 (- 20 1)}
29
```