Shawn Chen

Intro to C

Phase 2 PSI

1. Yup, Plenty. I've found myself in a loop of finding bugs, adding edge cases, getting lost, rewriting entire blocks etc.
2. I've had many double frees. I've had issues with (cons ()) lists and not properly passing ownership at one point.
3. I spent most of the allotted time rewriting phase 1 and everything there was memory safe with 0 lost blocks/double frees. Now that I've added a considerable amount of code for env and additional evaluator logic, I'm likely to have many issues. I struggle with ownership
4. I'm very proud of the evaluation logic. I was really struggling with it at first in phase 1 barely passing. I decided to rip it apart last minute and break it out into a few functions. That seemed to help a lot. Deciphering and deciding what I needed to back the env and just understanding how bindings/cells fit in was rewarding. I needed to go back and read the spec because initially I thought phase 2 wasn't going to be that bad...

0 psi> (+ (* 8 7) (- 6 3))

(+ WS (* WS 8 WS 7) WS (- WS 6 WS 3))

answer: pval_num: 59

1 psi> (* 3 4 (+ 5 6 7))s

(* WS 3 WS 4 WS (+ WS 5 WS 6 WS 7))

answer: pval_num: 216

2 psi> (-)

(-)

answer: pval_num: 0

3 psi> (/ 99 8 6)

(/ WS 99 WS 8 WS 6)

answer: pval_num: 2

4 psi> (* 0 3 4 +)

(* WS 0 WS 3 WS 4 WS +)

answer: $error{type-error: non-number in *}

5 psi> (* (+) 3 4 5)

(* WS (+) WS 3 WS 4 WS 5)

answer: pval_num: 0

6 psi> (= (- 4) -4)

(= WS (- WS 4) WS -4)

answer: pval_bool: #t

7 psi> (= 2.5 (/ 5 2))

(= WS 2.5 WS (/ WS 5 WS 2))

answer: nothing was here

8 psi> (= (= 4 5) (= 6 9))w

(= WS (= WS 4 WS 5) WS (= WS 6 WS 9))

answer: pval_bool: #t

9 psi> (= 3 4 (/ 1 0))

(= WS 3 WS 4 WS (/ WS 1 WS 0))

answer: $error{arithmetic-error: division by zero: 0}

0 psi> (/ 6 (if (= 3 4) 1 0))

(/ WS 6 WS (if WS (= WS 3 WS 4) WS 1 WS 0))

answer: $error{arithmetic-error: division by zero: 0}

1 psi> (/ 6 (if (= 3 3) 1 0))

(/ WS 6 WS (if WS (= WS 3 WS 3) WS 1 WS 0))

answer: pval_num: 6

2 psi> (/ 6 (if (= 3 3) 1))

(/ WS 6 WS (if WS (= WS 3 WS 3) WS 1))

answer: pval_num: 6

3 psi> (if (= 3 4) 7)

(if WS (= WS 3 WS 4) WS 7)

answer: pval_bool: #f

4 psi> ((if (= #f (= 0 #t)) + *) 6 (if (= #t #t) 7 8))

((if WS (= WS Bool(false) WS (= WS 0 WS Bool(true))) WS + WS *) WS 6 WS (if WS (= WS Bool(true) WS Bool(true)) WS 7 WS 8))

answer: pval_num: 13

5 psi> (if 0 1 2)

(if WS 0 WS 1 WS 2)

answer: $error{invalid-type: 0}

6 psi> (if () 4)

(if WS ( ) WS 4)

answer: $error{invalid-type}

7 psi> (if 0 1 b)

(if WS 0 WS 1 WS b)

answer: $error{invalid-type: 0}

8 psi> (if 1 2 (quit))

(if WS 1 WS 2 WS (quit))

answer: $error{invalid-type: 1}

9 psi> (quit)

(quit)

Quitting...

0 psi> a

a

answer: $error{unbound-var: a}

1 psi> (def a (= #t 4))

(def WS a WS (= WS Bool(true) WS 4))

answer: pval_bool: #f

2 psi> a

a

answer: pval_bool: #f

3 psi> (def a (* #t 4))

(def WS a WS (* WS Bool(true) WS 4))

answer: {type-error: non-number in *: #t}

4 psi> a

a

answer: pval_bool: #f

5 psi> (def b 5)

(def WS b WS 5)

answer: pval_num: 5

6 psi> (def c 6)

(def WS c WS 6)

answer: pval_num: 6

7 psi> (if a b c)

(if WS a WS b WS c)

answer: pval_num: 6

8 psi> (quit)

(quit)

Quitting...

0 psi> (quote 1)

(quote WS 1)

answer: pval_num: 1

1 psi> (quote)

(quote)

answer: $error{arity-error: quote}

2 psi> (quote d)

(quote WS d)

answer: pval_symbol: d

3 psi> (quote d e f)

(quote WS d WS e WS f)

answer: $error{arity-error: quote}

4 psi> (quote (d e f))

(quote WS (d WS e WS f))

answer: pval_symbol: d

pval_symbol: e

pval_symbol: f

5 psi> (+ 3 (quote 4))

(+ WS 3 WS (quote WS 4))

answer: pval_num: 7

6 psi> (def d 5)

(def WS d WS 5)

answer: pval_num: 5

7 psi> (= 5 (quote 5))

(= WS 5 WS (quote WS 5))

answer: pval_bool: #t

8 psi> (= d (quote 5))

(= WS d WS (quote WS 5))

answer: pval_bool: #t

9 psi> (= 5 (quote d))

(= WS 5 WS (quote WS d))

answer: pval_bool: #f

10 psi> (= d (quote d))

(= WS d WS (quote WS d))

answer: pval_bool: #f

11 psi> (def 5 e)

(def WS 5 WS e)

answer: $error{type-error: 5}0 psi> (def f '(1 2 3))

(def WS f WS quote (1 WS 2 WS 3))

answer: $error{unbound-var: quote}

1 psi> (cons 4 f)

(cons WS 4 WS f)

answer: $error{msg}

2 psi> (cons f 4)

(cons WS f WS 4)

answer: $error{value-error: 4}

3 psi> (cons 4 4)

(cons WS 4 WS 4)

answer: $error{value-error: 4}

4 psi> (cons () ())

(cons WS ( ) WS ( ))

answer:

5 psi> (head f)

(head WS f)

answer: $error{msg}

6 psi> (tail f)

(tail WS f)

answer: $error{msg}

7 psi> (tail (tail f))

(tail WS (tail WS f))

answer: $error{msg}

8 psi> (type (tail f))

(type WS (tail WS f))

answer: $error{unbound-var: type}

9 psi> (type (head f))

(type WS (head WS f))

answer: $error{unbound-var: type}

10 psi> (tail (tail (tail f)))

(tail WS (tail WS (tail WS f)))

answer: $error{msg}

11 psi> (quote ())

(quote WS ( ))

answer:

0 psi> (def h (* h 0))

(def WS h WS (* WS h WS 0))

answer: $error{unbound-var: h}

1 psi> (def h (cons 1 h))

(def WS h WS (cons WS 1 WS h))

answer: $error{msg}

2 psi> (def h (quote h))

(def WS h WS (quote WS h))

answer: pval_symbol: h

3 psi> (def h (cons 1 h))

(def WS h WS (cons WS 1 WS h))

answer: $error{value-error: h}

main.c

```c
#include <ctype.h>

#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/SinglyLinkedList.h"
#include "../include/operations.h"
#include "../include/pval.h"
#include "../include/token.h"
#include "../include/env.h"
#include "../include/cell.h"

pval *evaluate(pval *pval_list) {
    if (pval_list == NULL) {
        return NULL;
    }
    pval *res = _pval_evaluate(pval_list);
    pval_release(pval_list);
    return res;
}

pval *parse(token_stream *tk_stream) {
    if (tk_stream->count == 0) {
        return pval_bool(false);
    }
    pval *out;
    out =_parse_exp(0, tk_stream);

    // if((tk_stream->index != tk_stream->count - 1) && tk_stream->error == false){
    //     fprintf(stderr, "incomplete-parse\n");
    //     pval_release(out);
    //     return NULL;
    // }
    return out;

}

pval *_parse_exp(int idx, token_stream *tk_stream) {
    tk_stream->index = peek_next_token_idx(idx, tk_stream);
    token cur_tk = tk_stream->data[tk_stream->index];
```

```
    switch (cur_tk.type) {
        case TOK_INVALID:
            return NULL;
        case TOK_NUM:
            return pval_number(atoi(cur_tk.lexeme));
        case TOK_BOOL:
            if (strcmp("Bool(true)", cur_tk.lexeme) == 0) {
                return pval_bool(true);
            }
            return pval_bool(false);
        case TOK_COMMENT:
            return pval_str(cur_tk.lexeme);
        case TOK_SYM:
            return pval_symbol(cur_tk.lexeme);
        case TOK_LIST_L:
            // ++tk_stream->index;
            int peek_idx = peek_next_token_idx(++tk_stream->index, tk_stream);
            if(tk_stream->count == 1){
                return NULL;
            }
            pval *result = _parse_list(peek_idx, tk_stream);

            if (result != NULL) {
                return result;          //<============ func exit!
            }

            return NULL;
        case TOK_LIST_R:
            fprintf(stderr, "invalid-token: )\n");
            tk_stream->error = true;
            return NULL;
        default:
            if (cur_tk.type == TOK_WS) {
                return pval_bool(false);
            }
            return NULL;
    }
}

pval *_parse_list(int idx, token_stream *tk_stream) {
    pval *new_list = pval_empty_list();
    tk_stream->index = idx;

    while (tk_stream->index < tk_stream->count) {
        if (tk_stream->data[tk_stream->index].type == TOK_LIST_R) {
```

```
            return new_list;
        }
        pval *result = _parse_exp(tk_stream->index, tk_stream);

        if (result == NULL) {
            pval_release(new_list);
            return NULL;
            tk_stream->error = true;

        } else {
            append_node(new_list->value.val_list, create_node(result));
            result = pval_release(result);
            tk_stream->index = peek_next_token_idx(++tk_stream->index, tk_stream);
        }
    }
    pval_release(new_list);
    return NULL;
    tk_stream->error = true;
}

void interactive_input(void) {
    int cnt = 0;
    while (1)
      {
        printf("%d psi> ", cnt);
        token_stream *tk_stream = tokenize(stdin);

        print_tokens(tk_stream);

        printf("\n\n");

        pval *out = parse(tk_stream);
        // /* Ensure token stream is always freed */
        free_token_stream(tk_stream);

        out = evaluate(out);

        printf("answer: ");

        out = pval_print(out);
        printf("\n");

        pval_release(out);
        cnt ++;
    }
```

```
}

void debug_input(char *buf)
{
    printf("psi> ");

    token_stream *tk_stream = tokenize(fmemopen(buf, strlen(buf), "r"));
    print_tokens(tk_stream);

    printf("\n");

    pval *out = parse(tk_stream);
    // /* Ensure token stream is always freed */
    free_token_stream(tk_stream);



    out = evaluate(out);

    printf("answer: ");
    out = pval_print(out);

    out = pval_release(out);

    // if (out) {
    //     out = evaluate(out);
    //     if (out) {
    //         pval_print(out);
    //         out = pval_free(out);
    //     }
    //     // pval_free(out);
    // } else {
    //     printf("(no output)\n");
    // }

    // printf("\n\n");
}

int main() {
    init_global();
    char input[] = "";

    if (strcmp(input, "") != 0){

        debug_input(input);
```

```
    }
    else
    {
        interactive_input();
    }
    return 0;
}
```

Evaluator.c

```
#include <ctype.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/SinglyLinkedList.h"
#include "../include/cell.h"
#include "../include/env.h"
#include "../include/operations.h"
#include "../include/pval.h"
#include "../include/token.h"

char specials[6][6] = {"def", "if", "cons", "head", "tail", "quote"};


pval *eval_if(linked_node *node) {
    linked_node *n0 = node;
    linked_node *n1 = n0->next;
    linked_node *n2 = n1 ? n1->next : NULL;

    pval *a;
    pval *b;
    pval *c;

    if (!n1) {
        return pval_error(NULL, "arity-error");
    }

    if (((a = _pval_evaluate(n0->data))->type) != PVAL_BOOL) {
        return pval_error(a, "invalid-type");
    }
    if (a->value.val_bool) {
        b = _pval_evaluate(n1->data);

        return b;
```

```
    } else {
        if (n2 == NULL)
            return pval_bool(false);

        c = _pval_evaluate(n2->data);

        return c;
    }
}
pval *eval_def(linked_node *node) {
    binding *b;
    if (node->data->type != PVAL_SYMBOL) {
        return pval_error(node->data, "type-error");
    }
    if (!node->next) {
        return pval_error(node->data, "arity-error");
    }
    b = hash_find(env_global->table, node->data->value.val_symbol);

    if (b && b->is_protected) {
        return pval_error(node->data, "protected-symbol");
    }

    pval *in = _pval_evaluate(node->next->data);
    if (in->type == PVAL_ERROR)
        return in;

    if (b) {
        cell_set(b->cell, in);
        return in;
    } else {
        cell *c = cell_new(in);
        hash_probe_insert(env_global->table,
                          binding_new(node->data->value.val_symbol, c));
        cell_release(&c);
        return in;
    }
}

bool is_special_symbol(pval *pv) {
    for (int i = 0; i < 6; i++) {
        if (strcmp(specials[i], pv->value.val_symbol) == 0) {
            return true;
        }
    }
```

```
    return false;
}

pval *eval_special(pval *s, linked_node *node) {

    if (strcmp(s->value.val_symbol, "def") == 0) {
        return eval_def(node);
    } else if (strcmp(s->value.val_symbol, "if") == 0) {
        return eval_if(node);
    } else if (strcmp(s->value.val_symbol, "quote") == 0) {
        if (!node) {
            return pval_error(s, "arity-error");
        }
        if (node->next) {
            return pval_error(s, "arity-error");
        }

        pval *out = node->data;
        return pval_retain(out);
    } else if (strcmp(s->value.val_symbol, "cons") == 0){
        pval *head = node->data;
        pval *tail = node->next ? _pval_evaluate(node->next->data) : NULL;

        if (!head){
            return pval_error(NULL, "value-error");
        }
        else if(!tail){
            return pval_error(head, "value-error");
        }
        else if(tail->type != PVAL_LIST){
            //TODO: figure out how to print both elements
            return pval_error(tail, "value-error");
        }

        pval *list = pval_empty_list();

        pval *pv = pval_retain(head);
        append_node(list->value.val_list, create_node(pv));
        pv = pval_release(pv);

        linked_node *curr = tail->value.val_list->head;
        pval *tmp;

        while (curr) {
            tmp = pval_retain(curr->data);
```

```
            append_node(list->value.val_list, create_node(tmp));
            tmp = pval_release(tmp);
            curr = curr->next;
        }
        return pval_retain(list);
    } else if (strcmp(s->value.val_symbol, "head") == 0) {
        pval *lst = _pval_evaluate(node->data);
        if(lst->type != PVAL_LIST){
            return pval_error(lst, "value-error");
        }
        return lst->value.val_list->head->data;
    } else if (strcmp(s->value.val_symbol, "tail") == 0){
        pval *lst = _pval_evaluate(node->data);
        if(lst->type != PVAL_LIST){
            return pval_error(lst, "value-error");
        }
        linked_node *curr = lst->value.val_list->head->next;
        if(!curr){
            return lst->value.val_list->head->data;
        }
        pval *out = pval_empty_list();

        while(curr){
            append_node(out->value.val_list, create_node(curr->data));
            curr = curr->next;
        }
        return out;
    }


    else {
        return pval_error(NULL, "invalid-special-symbol?");
    }
}

pval *process_func(pval *func, pval *args){
    op_func funcptr = func->value.val_function;
    return funcptr(args);
}


pval *eval_symbol(pval *sym){
    binding *b = hash_find(env_global->table, sym->value.val_symbol);
    if(!b) return pval_error(sym,"unbound-var");
    return pval_retain(b->cell->value);
```

```
}

pval *eval_list(pval *list) {
    linked_node *node = list->value.val_list->head;
    pval *out;

    if (!node) {
        return pval_retain(list);
    }

    pval *head = node->data;

    if (head->type == PVAL_SYMBOL && is_special_symbol(head)) {
        out = eval_special(head, node->next);
        return out;
    }

    pval *func = _pval_evaluate(head);
    if (func == NULL) {
        return NULL;
    }
    if (func->type == PVAL_ERROR) {
        return func;
    }
    if (func->type == PVAL_LIST){
        return NULL;
    }
    if (func->type != PVAL_FUNCTION) {
        pval_release(func);
        return pval_error(NULL, "inapplicable-head");
    }

    pval *args_list = pval_empty_list();
    linked_node *curr = node->next;

    while (curr) {
        pval *arg_val = _pval_evaluate(curr->data);
        if (arg_val == NULL) {
            func = pval_release(func);
            args_list = pval_release(args_list);
            return NULL;
        }
        if (arg_val->type == PVAL_ERROR) {
            func = pval_release(func);
            args_list = pval_release(args_list);
```

```
            return arg_val;
        }
        append_node(args_list->value.val_list, create_node(arg_val));
        arg_val = pval_release(arg_val);
        curr = curr->next;
    }

    pval *result = process_func(func, args_list);
    func = pval_release(func);
    args_list = pval_release(args_list);

    return result;
}

pval *_pval_evaluate(pval *pv) {
    if (pv == NULL)
        return NULL;
    switch (pv->type) {
        case PVAL_NUMBER:
            return pval_retain(pv);
        case PVAL_BOOL:
            return pval_retain(pv);
        case PVAL_FUNCTION:
            return pval_retain(pv);
        case PVAL_ERROR:
            return pval_retain(pv);
        case PVAL_SYMBOL:
            return eval_symbol(pv);
        case PVAL_LIST:
            return eval_list(pv);
        default:
            break;
    }

    // Should never reach here
    return pval_error(pv, "unknown pval type");
}
```

Pval.c

```
#include <ctype.h>
#include <stdbool.h>
#include <stdint.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/SinglyLinkedList.h"
#include "../include/operations.h"
#include "../include/pval.h"
#include "../include/token.h"

pval *pval_retain(pval *pv){
    if(!pv) return NULL;
    pv->refcnt ++;
    return pv;
}

pval *pval_release(pval *pv){
    if(!pv) return NULL;
    if(pv->refcnt >= 1){
        pv->refcnt --;
    }
    if(pv->refcnt == 0){
        pv = pval_free(pv);
        return NULL;
    }
    return pv;
}

char *pval_type_to_string(pval *pv) {
    switch (pv->type) {
        case PVAL_NUMBER:
            return "PVAL_NUMBER";
        case PVAL_BOOL:
            return "PVAL_BOOL";
        case PVAL_SYMBOL:
            return "PVAL_SYMBOL";
        case PVAL_FUNCTION:
            return "PVAL_FUNCTION";
        case PVAL_LIST:
            return "PVAL_LIST";
        case PVAL_ERROR:
            return "PVAL_ERROR";
        default:
            return "UNKNOWN";
    }
}
```

```c
pval *pval_number(int64_t n) {
    pval *num = malloc(sizeof(*num));
    num->type = PVAL_NUMBER;
    num->value.val_number = n;
    num->refcnt = 1;
    return num;
}

pval *pval_bool(bool b) {
    pval *boolean = malloc(sizeof(*boolean));
    if (!boolean)
        return NULL;
    boolean->type = PVAL_BOOL;
    boolean->value.val_bool = b;
    boolean->refcnt = 1;

    return boolean;
}

pval *pval_symbol(char *symbol_buf) {
    pval *symbol = malloc(sizeof(*symbol));
    symbol->type = PVAL_SYMBOL;
    char *temp = strdup(symbol_buf);
    symbol->value.val_symbol = temp;
    symbol->refcnt = 1;

    return symbol;
}

pval *pval_error(pval *pv, char *msg) {
    pval *container = malloc(sizeof(*container));
    if (!container)
        return NULL;

    error *error = malloc(sizeof(*error));

    container->type = PVAL_ERROR;
    error->message = malloc(strlen(msg) + 1);
    if (pv) {
        error->pval_error = pv;
        pval_retain(pv);
    } else {
        error->pval_error = NULL;
    }
```

```
    strcpy(error->message, msg);
    container->value.val_error = error;
    container->refcnt = 1;
    return container;
}

pval *pval_empty_list() {
    pval *list = malloc(sizeof(*list));
    list->type = PVAL_LIST;
    list->value.val_list = create_list();
    list->refcnt = 1;
    return list;
}

pval *pval_str(char *str) {
    pval *str_pval = malloc(sizeof(*str_pval));
    str_pval->type = PVAL_STRING;
    str_pval->value.val_string = malloc(strlen(str) + 1);
    strcpy(str_pval->value.val_string, str);
    str_pval->refcnt = 1;
    return str_pval;
}

pval *pval_copy(pval *src) {
    pval *dst = malloc(sizeof(*dst));
    dst->type = src->type;
    dst->refcnt = 1;

    switch (src->type) {
        case PVAL_SYMBOL:
            dst->value.val_symbol = strdup(src->value.val_symbol);
            break;
        case PVAL_STRING:
            dst->value.val_string = strdup(src->value.val_string);
            break;
        case PVAL_LIST:
            pval *tmp = pval_list_handler(src, pval_copy, true);
            dst->value.val_list = tmp->value.val_list;
            free(tmp);
            break;
        case PVAL_ERROR:
            error *dst_error = malloc(sizeof(*dst_error));
            dst_error->message = strdup(src->value.val_error->message);

            if (src->value.val_error->pval_error != NULL) {
```

```
                dst_error->pval_error = pval_copy(src->value.val_error->pval_error);
            } else {
                dst_error->pval_error = NULL;
            }
            dst->value.val_error = dst_error;
            break;

        default:
            dst->value = src->value;  // primitives (number, bool, function pointer)
            break;
    }

    return dst;
}

pval *pval_free(pval *pv) {
    if (pv == NULL) {
        return NULL;
    }
    switch (pv->type) {
        case PVAL_SYMBOL:
            free(pv->value.val_symbol);
            pv->value.val_symbol = NULL;

            break;
        case PVAL_STRING:
            free(pv->value.val_string);
            pv->value.val_string = NULL;
            break;

        case PVAL_LIST:


            free_list(&(pv->value.val_list));
            break;

        case PVAL_ERROR:
            error *pv_err = pv->value.val_error;
            free(pv_err->message);
            if (pv_err->pval_error != NULL) {
                if (pv_err->pval_error->type == PVAL_SYMBOL) {
                    free(pv_err->pval_error->value.val_symbol);
                    free(pv_err->pval_error);
                } else if (pv_err->pval_error->type == PVAL_LIST) {
                    free_list(&pv_err->pval_error->value.val_list);
```

```
                }
                pv_err->message = NULL;
                pv_err->pval_error = NULL;
            }
            free(pv_err);
            pv_err = NULL;
            break;
        default:
            break;
    }
    free(pv);
    pv = NULL;
    return NULL;
}

/*
This function takes a CONTAINER of type PVAL_ERROR. the encapsulated pval within
error struct will never be passed.
*/

void pval_delete(pval *pv) {
    pval_free(pv);
}

pval *pval_print(pval *pv) {
    if (pv == NULL) {
        return NULL;
    }
    switch (pv->type) {
        case PVAL_NUMBER:
            printf("pval_num: %ld\n", pv->value.val_number);
            break;
        case PVAL_BOOL:
            printf("pval_bool: %s\n", pv->value.val_bool ? "#t" : "#f");
            break;
        case PVAL_STRING:
            printf("pval_string: %s\n", pv->value.val_string);
            break;
        case PVAL_SYMBOL:
            printf("pval_symbol: %s\n", pv->value.val_symbol);
            break;
        case PVAL_ERROR:
            pval_print_error(pv);
            break;
        case PVAL_LIST:
```

```
            pval_list_handler(pv, pval_print, false);
            break;
        default:
            printf("unknown_type: %s\n", pval_type_to_string(pv));
    }
    return pv;
}

pval *pval_list_handler(pval *pvals, pval *(*func)(pval *args), bool copy) {
    /*
    apply supplied function on whatever pvals in list;
    print pval*/
    if (pvals == NULL || pvals->type != PVAL_LIST) {
        return NULL;
    }
    pval *pval_list = copy ? pval_empty_list() : pvals;

    linked_node *curr_node = pvals->value.val_list->head;

    size_t idx = 0;
    while (idx < pvals->value.val_list->size) {
        if (curr_node->data->type == PVAL_LIST) {
            if (copy) {
                pval *nested_copy = pval_list_handler(curr_node->data, func, true);
                append_node(pval_list->value.val_list, create_node(nested_copy));
                nested_copy = NULL;
            } else {
                pval_list_handler(curr_node->data, func, false);
            }
        } else if (copy) {
            append_node(pval_list->value.val_list,
                        create_node(func(curr_node->data)));
        } else {
            func(curr_node->data);
        }
        curr_node = curr_node->next;
        idx++;
    }
    if (copy) {
        return pval_list;
    }
    return pvals;
}

// void pval_print_list_handler(pval *pvals) {
```

```
//     /*
//     if pvals is not list type:
//     print pval*/
//     if (pvals == NULL) {
//         return;
//     }
//     if (pvals->type != PVAL_LIST) {
//         return pval_print(pvals);
//     }
//     /*setup vars*/
//     linked_node *curr_node = pvals->value.val_list->head;

//     if (curr_node == NULL) {
//         printf("()");
//     } else {
//         while (curr_node) {
//             pval_print(curr_node->data);
//             curr_node = curr_node->next;
//         }
//     }
// }
void pval_print_error(pval *pv) {
    char *msg = pv->value.val_error->message;
    pval *offending_pval = pv->value.val_error->pval_error;
    if(pv->value.val_error->pval_error == NULL){
        printf("$error{%s}\n", msg);
        return;
    }
    switch (offending_pval->type) {
        case PVAL_SYMBOL:
            if (offending_pval->value.val_error->pval_error == NULL) {
                printf("{%s}", msg);
                break;
            }
            char *sym = offending_pval->value.val_symbol;
            printf("$error{%s: %s}\n", msg, sym);
            break;
        case PVAL_NUMBER:
            printf("$error{%s: %ld}\n", msg, offending_pval->value.val_number);
            break;
        case PVAL_BOOL:
            if (pv->value.val_bool == true) {
                printf("{%s: %s}\n", msg, "#t");
            } else {
                printf("{%s: %s}\n", msg, "#f");
```

```
            }
            break;
        case PVAL_FUNCTION:
            printf("$error{%s}\n", msg);
            break;
        case PVAL_LIST:
            printf("$error{%s}\n", msg);
            break;
        default:
            printf("$error{%s}\n", "msg");
    }
    msg = NULL;
    offending_pval = NULL;
}
```

Operations.c

```
#define _POSIX_C_SOURCE 200809L
#include "operations.h"

#include <math.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "SinglyLinkedList.h"

operation operations[] = {
    {"+", op_add},          {"-", op_sub},
    {"*", op_mul},          {"/", op_div},
    {"=", op_eq},           {"%", op_add},
    {"<", op_less},         {">", op_greater},
    {"!=", op_not},         {"&", op_add},
    {"$", op_add},          {"quit", op_quit},
    {"not", op_not},        {">=", op_greater_equal},
    {"<=", op_less_equal},
};

int op_len = sizeof(operations) / sizeof(operations[0]);

// this file defines the actual logic for the operations of + / * etc
```

```
pval *func_ptr_helper(op_func func){
    pval *fp = malloc(sizeof(*fp));
    fp->type = PVAL_FUNCTION;
    fp->value.val_function = func;
    fp->refcnt = 1;
    return fp;
}

pval *op_less_equal(pval *args){
    linked_node *curr;
    if (args->value.val_list->head == NULL) {
        return pval_bool(true);
    }
    curr = args->value.val_list->head;
    while (curr->next) {
        if (curr->data->type == curr->next->data->type &&
            curr->data->type == PVAL_NUMBER) {
            if (curr->data->value.val_number >
                curr->next->data->value.val_number) {
                return pval_bool(false);
            }
        }
        curr = curr->next;
    }
    return pval_bool(true);
}

pval *op_greater_equal(pval *args){
    linked_node *curr;
    if (args->value.val_list->head == NULL) {
        return pval_bool(true);
    }
    curr = args->value.val_list->head;
    while (curr->next) {
        if (curr->data->type == curr->next->data->type &&
            curr->data->type == PVAL_NUMBER) {
            if (curr->data->value.val_number <
                curr->next->data->value.val_number) {
                return pval_bool(false);
            }
        }
        curr = curr->next;
    }
    return pval_bool(true);
```

```
}

pval *op_not(pval *args){
    pval *pv = args->value.val_list->head->data;
    if(args->value.val_list->size != 1){
        return pval_error(NULL,"arity-error: 'not' requires 1 arg");
    } else if (pv->type == PVAL_BOOL && pv->value.val_bool == false){
        return pval_bool(true);
    } else {
        return pval_bool(false);
    }



}
pval *op_less(pval *args) {
    linked_node *curr;
    if (args->value.val_list->head == NULL) {
        return pval_bool(true);
    }
    curr = args->value.val_list->head;
    while (curr->next) {
        if (curr->data->type == curr->next->data->type &&
            curr->data->type == PVAL_NUMBER) {
            if (curr->data->value.val_number >=
                curr->next->data->value.val_number) {
                return pval_bool(false);
            }
        }
        curr = curr->next;
    }
    return pval_bool(true);
}
pval *op_greater(pval *args) {
    linked_node *curr;
    if (args->value.val_list->head == NULL) {
        return pval_bool(true);
    }
    curr = args->value.val_list->head;
    while (curr->next) {
        if (curr->data->type == curr->next->data->type &&
            curr->data->type == PVAL_NUMBER) {
            if (curr->data->value.val_number <=
                curr->next->data->value.val_number) {
                return pval_bool(false);
```

```
            }
        }
        curr = curr->next;
    }
    return pval_bool(true);
}



pval *op_add(pval *args) {
    if (args->type != PVAL_LIST)
        return pval_error(args, "type-error: expected list");

    int64_t result = 0;
    linked_node *curr = args->value.val_list->head;

    while (curr != NULL) {
        pval *elem = curr->data;
        if (elem->type != PVAL_NUMBER) {
            return pval_error(elem, "type-error: non-number in +");
        }
        result += elem->value.val_number;
        curr = curr->next;
    }

    return pval_number(result);
}

pval *op_sub(pval *args) {
    if (args->type != PVAL_LIST) {
        return pval_error(args, "type-error: expected list");
    }

    linked_node *curr = args->value.val_list->head;
    int64_t result = 0;

    while (curr != NULL) {
        pval *elem = curr->data;
        if (elem->type != PVAL_NUMBER) {
            return pval_error(elem, "type-error: non-number in -");
        }
        if (args->value.val_list->size == 1)  // make neg if single elem list
        {
            return pval_number(elem->value.val_number * -1);
        }
```

```
        if (elem == args->value.val_list->head->data) {
            result = elem->value.val_number;
            curr = curr->next;
            continue;
        }

        else {
            result -= elem->value.val_number;
        }
        curr = curr->next;
    }

    return pval_number(result);
}
pval *op_mul(pval *args) {
    if (args->type != PVAL_LIST) {
        return pval_error(args, "type-error: expected list");
    }

    linked_node *curr = args->value.val_list->head;
    int64_t prod = 0;

    while (curr != NULL) {
        pval *elem = curr->data;
        if (elem->type != PVAL_NUMBER) {
            return pval_error(elem, "type-error: non-number in *");
        }
        if (elem == args->value.val_list->head->data) {
            prod = elem->value.val_number;
        } else {
            prod *= elem->value.val_number;
        }
        curr = curr->next;
    }
    return pval_number(prod);
}

pval *op_div(pval *args) {
    if (args->type != PVAL_LIST) {
        return pval_error(args, "type-error: expected list");
    }

    if (args->value.val_list->size < 2) {
        return pval_error(args, "arity-error: requires >= 2 args");
    }
```

```
    linked_node *curr = args->value.val_list->head;
    float result = 0;

    while (curr != NULL) {
        pval *elem = curr->data;
        if (elem->type != PVAL_NUMBER) {
            return pval_error(elem, "type-error: non-number in /");
        }
        if (elem == args->value.val_list->head->data) {
            result = elem->value.val_number;
            if (!curr->next) {
                return pval_error(NULL, "arity-error: requires >= 2 args");
            }
        } else if (elem->value.val_number == 0) {
            return pval_error(elem, "arithmetic-error: division by zero");
        } else {
            result = (double)result / (double)elem->value.val_number;
            result = (int)floor(result);
        }
        curr = curr->next;
    }
    return pval_number(result);
}

pval *op_eq(pval *args) {
    if (args->type != PVAL_LIST)
        return pval_error(args, "type-error: expected list");

    linked_node *head = args->value.val_list->head;
    if (!head || !head->next)
        return pval_bool(true);

    pval *first = head->data;
    pval_type eq_type = first->type;
    linked_node *curr = head->next;

    while (curr) {
        pval *elem = curr->data;
        if (elem->type != eq_type)
            return pval_bool(false);

        switch (eq_type) {
            case PVAL_NUMBER:
                if (elem->value.val_number != first->value.val_number)
```

```
                    return pval_bool(false);
                break;
            case PVAL_BOOL:
                if (elem->value.val_bool != first->value.val_bool)
                    return pval_bool(false);
                break;
            case PVAL_SYMBOL:
                if (strcmp(elem->value.val_symbol, first->value.val_symbol) !=
                    0)
                    return pval_bool(false);
                break;
            default:
                return pval_error(elem, "type-error: unsupported type in =");
        }

        curr = curr->next;
    }

    return pval_bool(true);
}

pval *op_quit(pval *args) {
    (void)args;
    printf("Quitting...\n");
    exit(0);
}

pval *insert_op(pval *args) { //takes a symbol and converts into a func pval
    int op_index;
    if ((op_index = symbol_check(args->value.val_symbol)) != -1) {
        op_func func = operations[op_index].fn;
        free(args->value.val_symbol);
        args->value.val_symbol = NULL;
        args->type = PVAL_FUNCTION;
        args->value.val_function = func;
        return args;
    } else {
        return NULL;
    }
}

int symbol_check(char *in) {
    int op_len = sizeof(operations) / sizeof(operations[0]);
    for (int i = 0; i < op_len; i++) {
        if (strcmp(in, operations[i].name) == 0) {
```

```
            return i;
        }
    }
    return -1;
}
```

Token.c

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include "../include/token.h"
#include "../include/SinglyLinkedList.h"
#include "../include/operations.h"

token new_token(token_type type, char* lexeme){
    token tk;
    tk.type = type;

    if(lexeme != NULL){
        tk.lexeme = malloc(100);
        strcpy(tk.lexeme, lexeme);
    }
    else{
        tk.lexeme = NULL;
    }

    return tk;
}

void append_token(token_stream *tk_stream, token tok){
    if(tk_stream->count >= tk_stream->capacity){
        tk_stream->capacity *= 2;
        tk_stream->data = realloc(tk_stream->data, sizeof(token) * tk_stream->capacity);
        if(!tk_stream->data){
            fprintf(stderr, "realloc failed");
            exit(EXIT_FAILURE);
        }

    }
    tk_stream->data[tk_stream->count] = tok;
    tk_stream->count++;
}
token_stream *new_token_stream(){
```

```c
    token_stream *tk_stream = malloc(sizeof(*tk_stream));

    tk_stream->capacity = 16;
    tk_stream->data = malloc(sizeof(token) * tk_stream->capacity);
    tk_stream->count = 0;
    tk_stream->index = 0;
    tk_stream->error = false;

    return tk_stream;
}

int peek_next_token_idx(size_t idx, token_stream *tk_stream){
    while(idx < tk_stream->count){
        if(tk_stream->data[idx].type == TOK_WS){
            idx++;
            continue;
        }
        else{
            return idx;
        }
    }
    return -1;
}

void free_token_stream(token_stream *tk_stream){
    for(size_t i = 0; i < tk_stream->count; i++){
        free(tk_stream->data[i].lexeme);
    }
    free(tk_stream->data);
    free(tk_stream);

}

char buff_peek_next(char *buff, size_t idx){
    return buff[idx + 1];
}

char *num_token_helper(char *num_lexeme){
    size_t len = strlen(num_lexeme) + strlen("Number()") + 1;
    char *buf = malloc(len);
    snprintf(buf, len, "Number(%s)", num_lexeme);
    return buf;
}

char *symbol_token_helper(char *sym_lexeme){
```

```c
    size_t len = strlen(sym_lexeme) + strlen("Symbol(\"\")") + 1;
    char *buf = malloc(len);
    snprintf(buf, len, "Symbol(\"%s\")", sym_lexeme);
    return buf;
}

void null_term_helper(char *string, int len){
    string[len - 1] = '\0';
}

void print_tokens(token_stream *tk_stream){
    for(size_t i = 0; i < tk_stream->count; i++){
        printf("%s ", tk_stream->data[i].lexeme);
    }
}

void trim_new_line(token_stream *tk_stream){
    if(tk_stream->data[tk_stream->count - 1].type == TOK_WS){
        free(tk_stream->data[tk_stream->count - 1].lexeme);
        tk_stream->count--;
    }
}


token_stream *tokenize(FILE *fp){
    token_stream *tk_stream = new_token_stream();

    char stream_buf[4096];
    fgets(stream_buf, 4096, fp);
    printf("%s\n", stream_buf);

    int stream_idx = 0;
    char c = stream_buf[stream_idx];


    while(c != '\0'){
        switch(c){
            case ' ':// WS CHECK
                while (isspace(stream_buf[stream_idx]))
                {
                    stream_idx++;
                }
                append_token(tk_stream, new_token(TOK_WS, "WS"));
                break;
            case '\n': // WS CHECK - NEW LINE
```

```
            while (c == '\n')
            {
                c = stream_buf[++stream_idx];
            }
            append_token(tk_stream, new_token(TOK_WS, "WS"));
            break;
        case '\t': // WS CHECK - T
            while (isspace(stream_buf[stream_idx]))
            {
                stream_idx++;
            }
            append_token(tk_stream, new_token(TOK_WS, "WS"));
            break;
        case ';':// COMMENT
            char tmp_buff[100];
            int idx = 0;
            c = stream_buf[++stream_idx]; // skip ;
            while(c != '\0' && c != '\n'){
                tmp_buff[idx] = c;
                idx++;
                c = stream_buf[++stream_idx];
            }
            tmp_buff[idx] = '\0';
            append_token(tk_stream, new_token(TOK_COMMENT, tmp_buff));
            break;
        case '#':// BOOL
            if(buff_peek_next(stream_buf, stream_idx) == 't'){
                append_token(tk_stream, new_token(TOK_BOOL, "Bool(true)"));
            }
            else if (buff_peek_next(stream_buf, stream_idx) == 'f'){
                append_token(tk_stream, new_token(TOK_BOOL, "Bool(false)"));
            }
            else{
                char invalid[3] = {c, buff_peek_next(stream_buf, stream_idx), '\0'};
                append_token(tk_stream, new_token(TOK_INVALID, invalid));
                stream_idx = stream_idx + 2;
                break;
            }
            stream_idx = stream_idx + 2;
            c = stream_buf[stream_idx];
            break;
        case '\'':// SHORT HAND QUOTE
            append_token(tk_stream, new_token(TOK_SYM, "quote"));
            c = stream_buf[++stream_idx];
            break;
```

```
            case '(':// OPEN PAREN
                append_token(tk_stream, new_token(TOK_LIST_L, "("));
                c = stream_buf[++stream_idx];
                break;
            case ')':// CLOSE PAREN
                append_token(tk_stream, new_token(TOK_LIST_R, ")"));
                c = stream_buf[++stream_idx];
                break;
            default:
                if((c == '+' || c == '-') && isdigit(buff_peek_next(stream_buf, stream_idx))){ //NUM CHECK
WITH +/-



                    char tmp_buff[100];
                    tmp_buff[0] = c;
                    c = stream_buf[++stream_idx];
                    int idx = 1;
                    while(isdigit(c)){
                        tmp_buff[idx] = c;
                        idx++;
                        c = stream_buf[++stream_idx];
                    }
                    tmp_buff[idx] = '\0';



                    append_token(tk_stream, new_token(TOK_NUM, tmp_buff));
                    break;
                }
                else{
                    char tmp_buff[100];
                    int idx = 0;
                    bool is_num = true;

                    while (!isspace(c) && c != ')' && c != '(' && c != '\0') {
                        tmp_buff[idx] = c;
                        idx++;
                        if (!isdigit(c)) {
                            is_num = false;
                        }
                        c = stream_buf[++stream_idx];

                    }
                    tmp_buff[idx] = '\0';
                    char lead_chk[2] = {tmp_buff[0], '\0'};
```

```
                    if (is_num) {  // NUM CHECK NORMAL UNSIGNED
                        append_token(tk_stream, new_token(TOK_NUM, tmp_buff));
                        break;
                    } else {  // SYMBOL CHECK
                        if(symbol_check(tmp_buff) != -1){

                            append_token(tk_stream, new_token(TOK_SYM, tmp_buff));
                            break;
                        }
                        if (isdigit(tmp_buff[0]) || (symbol_check(lead_chk) != -1 && strlen(tmp_buff) >
1)) {

                            append_token(tk_stream, new_token(TOK_INVALID, tmp_buff));

                            break;
                        }
                        // if((symbol_check(lead_chk)) == -1 && strlen(tmp_buff)
                        // > 1) // COMMON SYMBOL - NOT QUIT
                        // {
                        //     append_token(tk_stream, new_token(TOK_INVALID,
                        //     tmp_buff)); break;
                        // }

                        append_token(tk_stream, new_token(TOK_SYM, tmp_buff));
                        break;
                    }
                }
        }
        c = stream_buf[stream_idx];
    }
    trim_new_line(tk_stream);
    return tk_stream;
}
```

Cell.c

```
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/pval.h"
#include "../include/cell.h"


binding *binding_new(char *name, cell *c){
```

```
    binding *b = malloc(sizeof(*b));
    b->name = strdup(name);
    cell_retain(c);
    b->cell = c;
    b->is_protected = false;
    return b;
}

void binding_free(binding **b){
    if(!b || !*b) return;
    free((*b)->name);
    cell_release(&((*b)->cell));
    free(*b);
    *b = NULL;
}

void cell_retain(cell *c) {
    if (!c)
        return;
    c->refcnt++;
}

void cell_release(cell **c) {
    if (!c || !*c)
        return;
    if ((*c)->refcnt <= 0)
        return;

    (*c)->refcnt--;

    if ((*c)->refcnt == 0) {
        pval_release((*c)->value);
        free(*c);
        *c = NULL;
    }
}
cell *cell_new(pval *pv){
    cell *out = malloc(sizeof(*out));
    out->value = pv;
    out->refcnt = 1;
    pval_retain(pv);
    return out;

}
void cell_set(cell *c, pval *pv){
```

```
    if (!c)
        return;

    if (c->value != pv) {
        pval_release(c->value);
        pval_retain(pv);
    }

    c->value = pv;
}
pval *cell_get(cell *c){
    return c->value;
}

void cell_free(cell *c){
    if (!c)
        return;
    pval_release(c->value);
    free(c);
}
```

Env.c

```
#include "../include/env.h"

#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/cell.h"
#include "../include/operations.h"
#include "../include/env.h"

// #include "../include/pval.h"


#define HASH_SIZE 256


static void _populate_built_ins(void){
    hashtable *ht = env_global->table;
    for(int i = 0; i < op_len; i++){
        pval *pv = func_ptr_helper(operations[i].fn);
```

```
        binding *b = binding_new(operations[i].name, cell_new(pv));
        b->is_protected = true;
        hash_probe_insert(ht, b);
    }
}

env *env_global = NULL;

env *init_global(void){
    env_global = malloc(sizeof(*env_global));
    env_global->parent = NULL;
    env_global->table = _init_hashtable();
    _populate_built_ins();
    return env_global;
}

env *env_new(env *parent){
    env *e = malloc(sizeof(*e));
    e->parent = parent;
    e->table = _init_hashtable();
    return e;
}
```

SinglyLinkedList.c

```
#include<stdio.h>
#include<stddef.h>
#include<stdlib.h>
#include "../include/pval.h"
#include "../include/SinglyLinkedList.h"

SinglyLinkedList *create_list(void) {
    SinglyLinkedList *list = malloc(sizeof(*list));
    list->head = NULL;
    list->tailp = &list->head;
    list->size = 0;
    return list;
}

void insert_node(SinglyLinkedList *list, linked_node *node) {
    node->next = list->head;
    list->head = node;
    list->size++;
    if (list->tailp == &list->head) {
```

```
        list->tailp = &node->next;
    }
}

void append_node(SinglyLinkedList *list, linked_node *node) {
    *list->tailp = node;
    list->tailp = &node->next;
    node->next = NULL;
    list->size++;
}

void remove_node(SinglyLinkedList *list, linked_node *node) {
    linked_node *curr, **pnp = &list->head;
    while ((curr = *pnp) != NULL) {
        if (curr == node) {
            *pnp = node->next;
            if (list->tailp == &node->next) {
                list->tailp = pnp;
            }
            list->size --;
            node->next = NULL;
            break;
        }
        pnp = &curr->next;
    }
}

void free_node(linked_node **n) {
    pval_release((*n)->data);
    free(*n);
    *n = NULL;
}

void free_list(SinglyLinkedList **list_ptr) {
    if(list_ptr == NULL || (*list_ptr) == NULL ) return;

    SinglyLinkedList *list = *list_ptr;

    linked_node *curr = list->head;
    while (curr) {
        linked_node *next = curr->next;
        free_node(&curr);
        curr = next;
    }
    free(list);
```

```
    *list_ptr = NULL;
}

linked_node *create_node(pval* data)
{
    linked_node *node = malloc(sizeof(*node));
    pval_retain(data);
    node->next = NULL;
    node->data = data;
    return node;
}
```

Hashtable.c

```
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/env.h"
#include "../include/cell.h"
// #include "../include/pval.h"

static binding *TOMBSTONE = (binding*)1;

#define HASH_SIZE 256


hashtable *_init_hashtable(void){
    hashtable *ht = malloc(sizeof(hashtable));
    ht->capacity = HASH_SIZE;
    ht->bindings = calloc(HASH_SIZE, sizeof(binding *)); // alllocate hash-size number of pointers. Using
calloc cuz it'll set 'em to null.
    return ht;
}

unsigned int hash(char *key){ //djb2
    size_t h = 5381;
    int c;

    while ((c = *key++)) {
        h = ((h << 5) + h) + c;
    }
    return h;
```

```
}

binding *hash_find(hashtable *ht, char *name) {
    int idx = hash(name) % ht->capacity;
    int offset = idx;

    while (true) {
        binding *b = ht->bindings[offset];
        if (b == NULL) {
            return NULL;
        } else if (b != TOMBSTONE && (strcmp(b->name, name)) == 0) {
            return b;
        }

        offset = (offset + 1) % ht->capacity;
    }
}

bool hash_probe_insert(hashtable *ht, binding *b) {

    int idx = (hash(b->name)) % ht->capacity;
    binding *bc;
    int offset = idx;
    int ts = -1;

    while (true) {
        bc = ht->bindings[offset];

        if (bc == NULL) {
            if(ts != -1){
                ht->bindings[ts] = b;
            } else{
                ht->bindings[offset] = b;
            }
            return true;

        } else if (bc == TOMBSTONE) {
            if (ts == -1) {
                ts = offset;
            }
        } else if (strcmp(bc->name, b->name) == 0) {
            cell_retain(b->cell);
            cell_release(&(bc->cell));
            bc->cell = b->cell;
            binding_free(&b);
```

```
            return true;
        }
        offset = (offset + 1) % ht->capacity;
    }

    return false;
}

bool hash_probe_delete(hashtable *ht, char *in){
    binding *bc;
    int offset = hash(in) % ht->capacity;

    while((bc = ht->bindings[offset]) != NULL){
        if (bc == TOMBSTONE) {
            offset = (offset + 1) % ht->capacity;
            continue;
        } else if(strcmp(bc->name, in) == 0){
            binding_free(&ht->bindings[offset]);
            ht->bindings[offset] = TOMBSTONE;
            return true;
        }
        offset = (offset + 1) % ht->capacity;
    }
    return false;
}
```