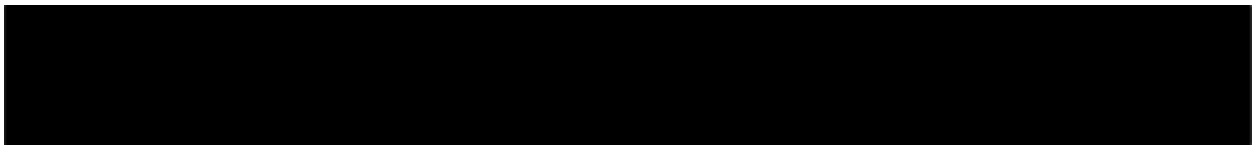
C48 Master Branch 10/26/17 7pm

Purpose
This document primarily aims to organize our functions in a small and easy to use database. The files in this document are located in the /src folder where most of our code is. Updating this document after a push is highly recommended, but not necessary. When updated, make sure to update the header.

Additions
Added a “Manual of Functions” that documents the comments in the code exactly as they appear. Found under our File Function Documentation.



File Function Documentation

Key

Struct
Enum
Function
Global Var

U - Uncommented/Needs better Documentation
H - Hard to Understand
F - Needs Formatting
D - Duplicate Function

---

compiler.c
2 Functions

	void compile()
	void assembler()

---

eval-apply.c
7 Functions

	int self_evaluatingp(object *exp)
	int primitivep(object *exp)

U	int variablep(object *exp)
	object *apply_primitive_procedure(object *procedure, object *arguments)
	char *apply(char operator, int arguments[])
U	object* lookup_variable_value(object *exp, object *env)
	char *eval(eval_arguments exp_env)

---

identifier.c
1 Struct, 2 Global Vars, 2 Functions

	struct identifier
U	static char *identifier_string
U	static double number_value
H	struct identifier *read_identifier(char *program, int index)
H	struct identifier *read_number(char *program, int index)

---

lexer.c
2 Functions

	token_list* lexer(char *program)
	token_list* list_lexer_tmp(char *program)

---

lexer_tmp.c
3 Functions

	token_list* lexer_tmp1(char *program)
U	token_list* lexer_tmp(char *program)
	token_list* list_lexer_tmp(char *program)

---

pair.c
5 Structs, 15 Functions

U	struct token

U	struct object
U	typedef struct token_list
U	token_list* create_token(struct token token, token_list *next)
U	token_list* prepend(struct token token, token_list *head)
U	int count_tokenlist(token_list *head)
U	token_list* reverse_tokenlist(token_list *head)
U	typedef struct pair_cell
U	typedef struct pair_token
U	void print(struct pair_token *list)
U	pair_cell* create1(void *car, void *cdr)
U	char* car(struct pair_token *list)
U	pair_token* cdr(struct pair_token* list)
U	pair_token* cons(void *car, pair_token *cdr)
U	pair_cell* cons1(struct object val, struct pair_cell *cdr)
U	int count_nodes1(pair_cell *head)
U	int count_nodes(pair_token *head)
U	pair_cell* reverse_code_tree(pair_cell *head)
U	pair_cell* remove_front(pair_cell *head)
U	pair_cell* read_from_tokens(struct pair_cell *token_list)

---

parser.c

1 Enum, 2 Structs, 11 Functions

	enum boolean
U	typedef struct object
H	object* cons(object *car, object *cdr)
H	object* car(object *cell)

H	object* cdr(object *cell)
	object* create_number(int number)
U	object* create_string(char *string)
	object* create_variable(char* variable)
U	object* create_boolean(char *variable)
U	object* create_primativeop(char* variable)
U	typedef struct type_list
	char* get_car(void *car)
	object* parse_rec(token_list *token_list, object *list_sofar)
	object* parse(token_list *token_list, object *code_tree)

---

print.c
1 Function

	char* print(object *result)

---

read.c
2 Structs, 2 Global Vars, 1 Function

U	static char *identifier_string
U	static double number_value
U	typedef struct eval_arguments_token
U	typedef struct eval_arguments_cell
U	struct eval_arguements1 parser(struct pair_cell *token_list)

---

read2.c
5 Global Vars, 1 Struct, 4 Functions

U	int left
U	int right

U	int invalid
U	int value
U	char charSet[]
U	struct Token
U	int isnumber(char s)
U	int isoperator(char s)
U	int isbrackets(char s)
U	char* read_token(char *program)

---

read_o.c
1 Enum, 2 Global Vars, 2 Structs, 7 Functions

U	enum Token
U	static char *identifier_string
U	static double number_value
U	typedef struct pair
U	typedef struct eval_arguements
U	pair* create1(void *car, void *cdr)
U	pair* cons(void *car, pair *cdr)
U	int isnumber(char *s)
U	struct eval_arguements read(char *program)
U	char* read_token(char *program)
U	int read_list(pair *list_so_far)
U	char* micro_read(char *program)

---

repl.c
Main

	int main(char *argc, char **argv[])

---

token.c

2 Structs, 12 Functions

	struct token_object
	typedef struct token_list
	char* token_type(char *token)
	token_list* reverse_token_list(token_list *head)
H	token_list* prepend_token(struct token_object val, struct token_list *cdr)
	int count_token_list(token_list *cursor)
U	char* first_value (struct token_list *list)
	char* first(struct token_list *list)
	char* find_value(struct token_list *list)
	char* find_type(struct token_list *list)
	token_list* rest(struct token_list *list)
	void print_token_list(struct token_list *list)
U	void print_token_list_value(struct token_list *list)
H	char* print_token_list_debug(struct token_list *list, char *result)

---

utils.c

6 Functions

	char* chopN(char *charBuffer, int n)
H	char* scat(char *s, char *t)
	int iswhitespace(char c)
	char* append(char *s, char c)
	int count_chars(char *string, char ch)
U	int our_isnumber(char *s)

---

vm.c

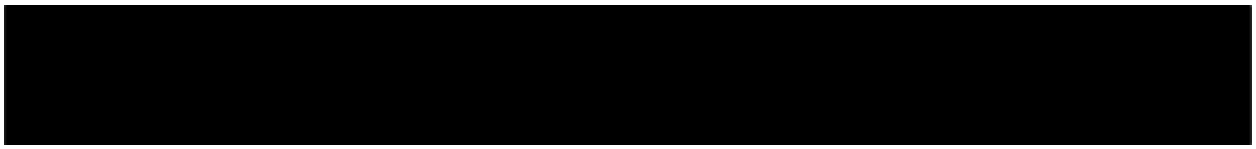
C48 Master Branch 10/26/17 7pm

3 Global Vars, 6 Functions

U	int MAXSIZE
U	int stack[8]
U	int top
U	int isEmpty()
U	int isFull()
U	int peek()
U	int pop()
U	int push(int data)
U	void machine(int code[])

Other files in src

	Makefile
	vm.h
	ztwild(Hello)



Manual of Functions

---

compiler.c

void compile()
Walk the abstract syntax tree and compile each expression.
void assembler()

Read the assembly file and write in bytes, use fread and fwrite.

---

eval-apply.c

int self_evaluatingp(object *exp)
This function determines whether the given argument is a number or string Parameters: - exp, this is an object containing an argument value Return Value: - 1 if the given object is a number or a string - 0 otherwise
int primitivep(object *exp)
This function determines whether a given expression is primitive Parameters: - exp, this is an object containing an expression Return Value: - 1 if the given object is a primitive expression - 0 otherwise
int variablep(object *exp)
object *apply_primitive_procedure(object *procedure, object *arguments)
This function executes our primitive operations i.e. +, -, *, / Parameters: - procedure, - arguments, Return Value: - procedure,
char *apply(char operator, int arguments[])
Executes the operation returning the result This function implements our defined functions among other operation native to c Parameters: - operator the function or operation to perform - arguments[] an array containing the necessary arguments to execute the command Return Value:

- answer the result of the operation Example call: apply('+', int[1,2]) expected return val: 3
object* lookup_variable_value(object *exp, object *env)
char *eval(eval_arguments exp_env)
Evaluates the given arguments to create the code tree that apply will execute over This function takes in an expression and and the environment hash table and creates the tree Parameters: - exp_env the expression and the environment the expression is in Return Value: - answer the head of the code tree Example call: expected return val:

---

identifier.c

struct identifier
identifier is Parameters: - identifier_token, used to determine the type of identifier - length, the length of the identifier plus whitespace
static char *identifier_string
static double number_value
struct identifier *read_identifier(char *program, int index)
This function reads identifiers from the given program Parameters: - program the string containing the code we will run - the index of the identifier to read

Return Value: - ??? Example call: expected return val:
struct identifier *read_number(char *program, int index)
This function reads numbers from the given program Parameters: - program the string containing the code we will run - the index of the number to read Return Value: - ??? Example call: expected return val:

---

lexer.c

token_list* lexer(char *program)
This function creates the preliminary tree The returned tree is generated by iterating through program and creating tokens for each statement Parameters: - program the string containing the code we will run Return Value: - a list of tokens created by iterating through program Example call: List_Lexer("(+ 1 1)"); expected return val: tokenlist of length 5
token_list* list_lexer_tmp(char *program)
This is a temp version of list_lexer to test the creation of tokens This function creates the preliminary tree The returned tree is generated by iterating through program and creating tokens for each statement Parameters: - program the string containing the code we will run Return Value: - a list of tokens created by iterating through program Example call: List_Lexer_tmp("(+ 1 1)"); expected return val: tokenlist of length 5

---

lexer_tmp.c

token_list* lexer_tmp1(char *program)
This function creates the preliminary tree The returned tree is generated by iterating through program and creating tokens for each statement Parameters: - program the string containing the code we will run Return Value: - a list of tokens created by iterating through program Example call: List_Lexer("(+ 1 1)"); expected return val: tokenlist of length 5
token_list* lexer_tmp(char *program)
token_list* list_lexer_tmp(char *program)
This is a temp version of list_lexer to test the creation of tokens This function creates the preliminary tree The returned tree is generated by iterating through program and creating tokens for each statement Parameters: - program the string containing the code we will run Return Value: - a list of tokens created by iterating through program Example call: List_Lexer_tmp("(+ 1 1)"); expected return val: tokenlist of length 5

---

pair.c

struct token
struct object
typedef struct token_list

token_list* create_token(struct token token, token_list *next)
token_list* prepend(struct token token, token_list *head)
int count_tokenlist(token_list *head)
token_list* reverse_tokenlist(token_list *head)
typedef struct pair_cell
typedef struct pair_token
void print(struct pair_token *list)
pair_cell* create1(void *car, void *cdr)
char* car(struct pair_token *list)
pair_token* cdr(struct pair_token* list)
pair_token* cons(void *car, pair_token *cdr)
pair_cell* cons1(struct object val, struct pair_cell *cdr)
int count_nodes1(pair_cell *head)

int count_nodes(pair_token *head)
pair_cell* reverse_code_tree(pair_cell *head)
pair_cell* remove_front(pair_cell *head)
pair_cell* read_from_tokens(struct pair_cell *token_list)

---

parser.c

typedef struct object
object* cons(object *car, object *cdr)
This function constructs ???? Parameters: - car, the first object in the code tree - cdr, the rest of the objects in the code tree Return Value: - test1, an object ???
object* car(object *cell)
This function checks if the cells car is ??? Parameters: - cell, an object ??? Return Value: - ???
object* cdr(object *cell)
This function checks that the cells cdr is ??? Parameters: - cell Return Value: - ???
object* create_number(int number)

This function creates an object representing a number Parameters: - number, the value of the object to create Return Value: - num an object with type number and value number
object* create_string(char* variable)
object* create_variable(char* variable)
This function creates an object representing a variable Parameters: - variable the value of the object to create Return Value: -var, an object with type "variable", and value variable
object* create_boolean(char* variable)
object* create_primativeop(char* variable)
This function creates an object representing a primitive operation Parameters: - variable the value of the object to create Return Value: - var, an object with type "primeop", and value variable
typedef struct type_list
char* get_car(void *car)
Use this when you dereference void pointer to pointer fine This function gets the head of the code tree Parameters: -car, the cell containing the the type and value of the head of the code tree Return Value: -car, a c string of the value of the head of the code_tree
object* parse_rec(token_list *token_list, object *code_tree)
Recursively adds tokens to the code_tree Parameters: -token_list, the list of tokens to add to the code_tree -code_tree, the code_tree being built

object* parse(token_list *token_list, object *code_tree)

---

print.c

char* print(object *result)
This function prints the given object Parameters: -result, the object to print

---

read.c

static char *identifier_string
static double number_value
typedef struct eval_arguments_token
typedef struct eval_arguments_cell
struct eval_arguements1 parser(struct pair_cell *token_list)

---

read2.c

int left
int right

int invalid
int value
char charSet[]
struct Token
int isnumber(char s)
int isoperator(char s)
int isbrackets(char s)
char* read_token(char *program)

---

read_o.c

enum Token
static char *identifier_string
static double number_value
typedef struct pair

typedef struct eval_arguements
pair* create1(void *car, void *cdr)
pair* cons(void *car, pair *cdr)
int isnumber(char *s)
struct eval_arguements read(char *program)
char* read_token(char *program)
int read_list(pair *list_so_far)
char* micro_read(char *program)

---

repl.c

int main(char *argc, char **argv[])

---

token.c

struct token_object
token_objects is represent the tokens in our token list Parameters: - type, a character representing whether the token is an identifier, operator, or an argument - value, the original string value of the token. i.e. 5, (, +, etc.

typedef struct token_list
token_list is a collection of token_objects used to create a code tree Parameters: - val, the value of the token_object in the list - next, points to the next token_object in the list
char* token_type(char *token)
This function determines the type used in the creation of a token Parameters: - token the string we are creating the token from Return Value: - the type the token has, identifier, num, or symbol. Example call: token_type('+'); expected return val: identifier
token_list* reverse_token_list(token_list *head)
This function recursively constructs a the token list in reverse order Parameters: - head the start of the token list to reverse Return Value: - head the start of the reversed token list
token_list* prepend_token(struct token_object val, struct token_list *cdr)
This function does ??? Parameters: - val ??? - cdr the rest of the tokenlist Return Value: - pair Example call: expected return val:
int count_token_list(token_list *cursor)
This function counts the rest of the elements in a tokenlist Parameters: - cursor the list of tokens to count Return Value: - c the number of tokens in the tokenlist
char* first_value (struct token_list *list)

char* first(struct token_list *list)
This function returns the value of the first element in a list Parameters: - list the list you want the head of 1 Return Value: - the value of the first element in the list Example call: expected return val:
char* find_value(struct token_list *list)
This function returns the type of the current token from the given token_list Parameters: - list, the head of the token_list Return Value: - the value of the head token from list - 0 if list in NULL
char* find_type(struct token_list *list)
This function determines the type of the current token from the token_list Parameters: - list, the head of the token_list Return Value: - character representing the value of the token from list or 0 if list is NULL
token_list* rest(struct token_list *list)
This function returns the element in the next position in a list Parameters: - list the list you are calling rest on Return Value: - the next element in the list Example call: expected return val:
void print_token_list(struct token_list *list)
This function recursively prints the token list Parameters: - list the token to print - result the tokenlist to print from

Return Value: - result returns the list of tokens after printing them Example call: expected return val:
void print_token_list_value(struct token_list *list)
char* print_token_list_debug(struct token_list *list, char *result)
This is a testing function to test the ability to read from and utilize our token_list this function is recursive Parameters: - list, the token_list we are testing over - result, ??? Return Value: - result, used to recurse through our list

---

utils.c

char* chopN(char *charBuffer, int n)
This function advance the pointer to a new location effectivly removing N character from the string Parameters: - the buffer to edit - n the number of characters to chop off Return Value: - charBuffer with N characters chopped off
char* scat(char *s, char *t)
This function does ??? https://stackoverflow.com/questions/14259540/c-programming-strcat-using-pointer Parameters: - s ??? - t ??? Return Value: - p ??? Example call: expected return val:
int iswhitespace(char c)

This function checks if a given character is whitespace Parameters: - c the character to check Return Value: - 1 if the character is whitespace - 0 otherwise
char* append(char *s, char c)
This function appends a given char to the end of the given string Parameters: - s the string to append characters to - c the char to append Return Value: - a string with the given char at the end of the original string Example call: append("hello", '!'); expected return val: hello!
int count_chars(char *string, char ch)
This function calculates the length of the given string We are computing the length once at this point because it is a relatively lengthy operation, and we don't want to have to compute it anew every time the i < length condition is checked. Parameters: - program the string containing the code we will run Return Value: - a list of tokens created by iterating through program Example call: List_Lexer_tmp("(+ 1 1)"); expected return val: tokenlist of length 5
int our_isnumber(char *s)

---

vm.c

int MAXSIZE
int stack[8]

C48 Master Branch 10/26/17 7pm

int top
int isEmpty()
int isFull()
int peek()
int pop()
int push(int data)
void machine(int code[])