***env.h***

#ifndef \_MAL\_ENV\_H

#define \_MAL\_ENV\_H

#include "libs/linked\_list/linked\_list.h"

#include "libs/hashmap/hashmap.h"

#include "types.h"

typedef struct Env\_s Env;

struct Env\_s {

struct Env\_s\* outer;

hashmap data;

};

Env\* env\_make(Env\* outer, list binds, list exprs, MalType\* variadic\_symbol);

Env\* env\_set(Env\* current, MalType\* symbol, MalType\* value);

Env\* env\_set\_C\_fn(Env\* current, char\* symbol\_name, MalType\*(\*fn)(list));

MalType\* env\_get(Env\* current, MalType\* symbol);

Env\* env\_find(Env\* current, MalType\* symbol);

#endif

***printer.h***

#ifndef \_PRINTER\_H

#define \_PRINTER\_H

#include <stdarg.h>

#include "types.h"

#define UNREADABLY 0

#define READABLY 1

char\* pr\_str(MalType\* mal\_val, int readably);

char\* pr\_str\_list(list lst, int readably, char\* start\_delimiter, char\* end\_delimiter, char\* separator);

char\* escape\_string(char\* str);

char\* snprintfbuf(long initial\_size, char\* fmt, ...);

#endif

***step4\_if\_fn\_do.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#include "core.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define SYMBOL\_IF "if"

#define SYMBOL\_FNSTAR "fn\*"

#define SYMBOL\_DO "do"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\* env);

MalType\* eval\_letstar(MalType\* ast, Env\* env);

MalType\* eval\_if(MalType\* ast, Env\* env);

MalType\* eval\_fnstar(MalType\* ast, Env\* env);

MalType\* eval\_do(MalType\* ast, Env\* env);

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

return eval\_letstar(ast, env);

}

else if (strcmp(symbol, SYMBOL\_IF) == 0) {

return eval\_if(ast, env);

}

else if (strcmp(symbol, SYMBOL\_FNSTAR) == 0) {

return eval\_fnstar(ast, env);

}

else if (strcmp(symbol, SYMBOL\_DO) == 0) {

return eval\_do(ast, env);

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else if (is\_closure(func)) {

MalClosure\* closure = func->value.mal\_closure;

list params = (closure->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(evlst->next);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !closure->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

Env\* new\_env = env\_make(closure->env, params, evlst->next, closure->more\_symbol);

return EVAL(closure->definition, new\_env);

}

}

else {

return make\_error\_fmt("Error: first item in list is not callable: %s.", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

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

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.4\n");

puts("Press Ctrl+d to exit\n");

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

ns\* core = ns\_make\_core();

hashmap mappings = core->mappings;

while (mappings) {

char\* symbol = mappings->data;

MalType\*(\*function)(list) = (MalType\*(\*)(list))mappings->next->data;

env\_set\_C\_fn(repl\_env, symbol, function);

/\* pop symbol and function from hashmap/list \*/

mappings = mappings->next->next;

}

/\* add not function \*/

/\* not using rep as it prints the result \*/

EVAL(READ("(def! not (fn\* (a) (if a false true)))"), repl\_env);

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'def!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'def!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, env);

if (!is\_error(result)){

env = env\_set(env, defbang\_symbol, result);

}

return result;

}

MalType\* eval\_letstar(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'let\*': missing bindings list");

}

MalType\* bindings = lst->next->data;

MalType\* forms = lst->next->next ? lst->next->next->data : make\_nil();

if (!is\_sequential(bindings)) {

return make\_error("'let\*': first argument is not list or vector");

}

list bindings\_list = bindings->value.mal\_list;

if (list\_count(bindings\_list) % 2 == 1) {

return make\_error("'let\*': expected an even number of binding pairs");

}

Env\* letstar\_env = env\_make(env, NULL, NULL, NULL);

/\* evaluate the bindings \*/

while(bindings\_list) {

MalType\* symbol = bindings\_list->data;

MalType\* value = EVAL(bindings\_list->next->data, letstar\_env);

/\* early return from error \*/

if (is\_error(value)) { return value; }

env\_set(letstar\_env, symbol, value);

bindings\_list = bindings\_list->next->next;

}

return EVAL(forms, letstar\_env);

}

MalType\* eval\_if(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

if (!lst->next || !lst->next->next) {

return make\_error("'if': too few arguments");

}

if (lst->next->next->next && lst->next->next->next->next) {

return make\_error("'if': too many arguments");

}

MalType\* condition = EVAL(lst->next->data, env);

if (is\_error(condition)) { return condition; }

if (is\_false(condition) || is\_nil(condition)) {

/\* check whether false branch is present \*/

if (lst->next->next->next) {

return EVAL(lst->next->next->next->data, env);

}

else {

return make\_nil();

}

} else {

return EVAL(lst->next->next->data, env);

}

}

MalType\* eval\_fnstar(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* regularise\_parameters(list\* params, MalType\*\* more);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'fn\*': missing argument list");

}

else if (!lst->next->next) {

return make\_error("'fn\*': missing function body");

}

MalType\* params = lst->next->data;

list params\_list = params->value.mal\_list;

MalType\* more\_symbol = NULL;

MalType\* result = regularise\_parameters(&params\_list, &more\_symbol);

if (is\_error(result)) { return result; }

MalType\* definition = lst->next->next->data;

MalType\* regular\_params = make\_list(params\_list);

return make\_closure(env, regular\_params, definition, more\_symbol);

}

MalType\* eval\_do(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

/\* handle empty 'do' \*/

if (!lst->next) { return make\_nil(); }

/\* evaluate all but the last form \*/

lst = lst->next;

while (lst->next) {

MalType\* val = EVAL(lst->data, env);

/\* return error early \*/

if (is\_error(val)) { return val; }

lst = lst->next;

}

/\* return the last value \*/

return EVAL(lst->data, env);

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* regularise\_parameters(list\* args, MalType\*\* more\_symbol) {

/\* forward reference \*/

char\* symbol\_fn(gptr data);

list regular\_args = NULL;

while (\*args) {

MalType\* val = (\*args)->data;

if (!is\_symbol(val)) {

return make\_error\_fmt("non-symbol found in fn argument list '%s'", \

pr\_str(val, UNREADABLY));

}

if (val->value.mal\_symbol[0] == '&') {

/\* & is found but there is no symbol \*/

if (val->value.mal\_symbol[1] == '\0' && !(\*args)->next) {

return make\_error("missing symbol after '&' in argument list");

}

/\* & is found and there is a single symbol after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next &&

is\_symbol((\*args)->next->data) && !(\*args)->next->next)) {

/\* TODO: check symbol is no a duplicate of one already on the list \*/

\*more\_symbol = (\*args)->next->data;

break;

}

/\* & is found and there extra symbols after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next && (\*args)->next->next)) {

return make\_error\_fmt("unexpected symbol after '& %s' in argument list: '%s'", \

pr\_str((\*args)->next->data, UNREADABLY), \

pr\_str((\*args)->next->next->data, UNREADABLY));

}

/\* & is found as part of the symbol and no other symbols \*/

else if (val->value.mal\_symbol[1] != '\0' && !(\*args)->next) {

\*more\_symbol = make\_symbol((val->value.mal\_symbol + 1));

break;

}

/\* & is found as part of the symbol but there are other symbols after \*/

else if (val->value.mal\_symbol[1] != '\0' && (\*args)->next) {

return make\_error\_fmt("unexpected symbol after '%s' in argument list: '%s'", \

pr\_str(val, UNREADABLY), \

pr\_str((\*args)->next->data, UNREADABLY));

}

}

/\* & is not found - add the symbol to the regular argument list \*/

else {

if (list\_findf(regular\_args, val->value.mal\_symbol, symbol\_fn) > 0) {

return make\_error\_fmt("duplicate symbol in argument list: '%s'", pr\_str(val, UNREADABLY));

}

else {

regular\_args = list\_push(regular\_args, val);

}

}

\*args = (\*args)->next;

}

\*args = list\_reverse(regular\_args);

return make\_nil();

}

char\* symbol\_fn(gptr data) {

MalType\* val = data;

return (val->value.mal\_symbol);

}

/\* silence the compiler after swap!, apply, and map are added to the core \*/

MalType\* apply(MalType\* ast, Env\* env) {

return make\_nil();

}

***reader.h***

#ifndef \_MAL\_READER\_H

#define \_MAL\_READER\_H

#include "types.h"

typedef struct Token\_s {

int type;

char\* data;

char\* error;

} Token;

typedef struct Reader\_s {

long position; // current position in the array

long token\_count; // number of tokens in the array

long max\_tokens; // maximum number of tokens the array can hold

Token\*\* token\_data; // pointer to an array of Tokens

char\* error; // error message

} Reader;

/\* reader object \*/

Reader\* reader\_make(long token\_capacity);

Reader\* reader\_append(Reader\* reader, Token\* token);

Token\* reader\_peek(const Reader\* reader);

Token\* reader\_next(Reader\* reader);

Token\* reader\_get\_at(const Reader\* reader, long i);

void reader\_print(Reader\* reader);

/\* tokenizing the input \*/

Reader\* tokenize(char\* token\_string);

char\* read\_fixed\_length\_token(char\* current, Token\*\* ptoken, int n);

char\* read\_string\_token(char\* current, Token\*\* ptoken);

char\* read\_comment\_token(char\* current, Token\*\* ptoken);

//char\* read\_integer\_token(char\* current, Token\*\* ptoken);

char\* read\_number\_token(char\* current, Token\*\* ptoken);

char\* read\_symbol\_token(char\* current, Token\*\* ptoken);

char\* read\_keyword\_token(char\* current, Token\*\* ptoken);

/\* reading the tokens into types \*/

MalType\* read\_str(char\* token\_string);

MalType\* read\_form(Reader\* reader);

MalType\* read\_atom(Reader\* reader);

MalType\* read\_list(Reader\* reader);

MalType\* read\_vector(Reader\* reader);

MalType\* read\_hashmap(Reader\* reader);

/\* utility functions \*/

char\* read\_terminated\_token (char\* current, Token\*\* ptoken, int type);

MalType\* read\_matched\_delimiters(Reader\* reader, char start\_delimiter, char end\_delimiter);

MalType\* make\_symbol\_list(Reader\* reader, char\* symbol\_name);

Token\* token\_allocate(char\* str, long num\_chars, int type, char\* error);

char\* unescape\_string(char\* str, long length);

#endif

***core.h***

#ifndef \_MAL\_CORE\_H

#define \_MAL\_CORE\_H

#include "libs/hashmap/hashmap.h"

#include "types.h"

typedef struct ns\_s ns;

struct ns\_s {

hashmap mappings;

};

ns\* ns\_make\_core();

MalType\* as\_str(list args, int readably, char\* separator);

MalType\* print(list args, int readably, char\* separator);

char\* get\_fn(gptr data);

MalType\* equal\_lists(MalType\* lst1, MalType\* lst2);

MalType\* equal\_hashmaps(MalType\* map1, MalType\* map2);

#endif

***types.h***

#ifndef \_MAL\_TYPES\_H

#define \_MAL\_TYPES\_H

#include "libs/linked\_list/linked\_list.h"

#include "libs/hashmap/hashmap.h"

#define MALTYPE\_SYMBOL 1

#define MALTYPE\_KEYWORD 2

#define MALTYPE\_INTEGER 3

#define MALTYPE\_FLOAT 4

#define MALTYPE\_STRING 5

#define MALTYPE\_TRUE 6

#define MALTYPE\_FALSE 7

#define MALTYPE\_NIL 8

#define MALTYPE\_LIST 9

#define MALTYPE\_VECTOR 10

#define MALTYPE\_HASHMAP 11

#define MALTYPE\_FUNCTION 12

#define MALTYPE\_CLOSURE 13

#define MALTYPE\_ERROR 14

#define MALTYPE\_ATOM 15

typedef struct MalType\_s MalType;

typedef struct MalClosure\_s MalClosure;

typedef struct Env\_s Env;

struct MalType\_s {

int type;

int is\_macro;

MalType\* metadata;

union MalValue {

long mal\_integer;

double mal\_float;

char\* mal\_symbol;

char\* mal\_string;

char\* mal\_keyword;

list mal\_list;

/\* vector mal\_vector; TODO: implement a real vector \*/

/\* hashmap mal\_hashmap; TODO: implement a real hashmap \*/

MalType\* (\*mal\_function)(list);

MalClosure\* mal\_closure;

MalType\* mal\_atom;

MalType\* mal\_error;

} value;

};

struct MalClosure\_s {

Env\* env;

MalType\* parameters;

MalType\* more\_symbol;

MalType\* definition;

};

MalType\* make\_symbol(char\* value);

MalType\* make\_integer(long value);

MalType\* make\_float(double value);

MalType\* make\_keyword(char\* value);

MalType\* make\_string(char\* value);

MalType\* make\_list(list value);

MalType\* make\_vector(list value);

MalType\* make\_hashmap(list value);

MalType\* make\_true();

MalType\* make\_false();

MalType\* make\_nil();

MalType\* make\_atom(MalType\* value);

MalType\* make\_error(char\* msg);

MalType\* make\_error\_fmt(char\* fmt, ...);

MalType\* wrap\_error(MalType\* value);

MalType\* make\_function(MalType\*(\*fn)(list args));

MalType\* make\_closure(Env\* env, MalType\* parameters, MalType\* definition, MalType\* more\_symbol);

MalType\* copy\_type(MalType\* value);

int is\_sequential(MalType\* val);

int is\_self\_evaluating(MalType\* val);

int is\_list(MalType\* val);

int is\_vector(MalType\* val);

int is\_hashmap(MalType\* val);

int is\_nil(MalType\* val);

int is\_string(MalType\* val);

int is\_integer(MalType\* val);

int is\_float(MalType\* val);

int is\_number(MalType\* val);

int is\_true(MalType\* val);

int is\_false(MalType\* val);

int is\_symbol(MalType\* val);

int is\_keyword(MalType\* val);

int is\_atom(MalType\* val);

int is\_error(MalType\* val);

int is\_callable(MalType\* val);

int is\_function(MalType\* val);

int is\_closure(MalType\* val);

int is\_macro(MalType\* val);

#endif

***Dockerfile***

FROM ubuntu:bionic

MAINTAINER Duncan Watts <fungiblecog@gmail.com>

##########################################################

# General requirements for testing or common across many

# implementations

##########################################################

RUN apt-get -y update

# Required for running tests

RUN apt-get -y install make python

# Some typical implementation and test requirements

#RUN apt-get -y install curl

RUN mkdir -p /mal

WORKDIR /mal

##########################################################

# Specific implementation requirements

##########################################################

# Install gcc

RUN apt-get -y install gcc

# Libraries needed for the C impl

RUN apt-get -y install libffi-dev libgc-dev libedit-dev

***Makefile***

CC = gcc

CFLAGS = -std=c99 -g -Wall

LIBS = -ledit -lgc

FFI\_LIBS = -ldl -lffi

SRC = reader.c printer.c types.c env.c core.c

HEADERS = reader.h printer.h types.h env.h core.h

LIB\_DIR = ./libs

LIB\_LIST\_H = $(LIB\_DIR)/linked\_list/linked\_list.h

LIB\_LIST\_SRC = $(LIB\_DIR)/linked\_list/linked\_list.c

LIB\_MAP\_H = $(LIB\_DIR)/hashmap/hashmap.h

LIB\_MAP\_SRC = $(LIB\_DIR)/hashmap/hashmap.c

LIBS\_H = $(LIB\_LIST\_H) $(LIB\_MAP\_H)

LIBS\_SRC = $(LIB\_LIST\_SRC) $(LIB\_MAP\_SRC)

S0\_SRC = step0\_repl.c

S1\_SRC = step1\_read\_print.c reader.c types.c printer.c $(LIB\_LIST\_SRC)

S2\_SRC = step2\_eval.c reader.c types.c printer.c $(LIBS\_SRC)

S3\_SRC = step3\_env.c reader.c types.c printer.c env.c $(LIBS\_SRC)

S4\_SRC = step4\_if\_fn\_do.c $(SRC) $(LIBS\_SRC)

S5\_SRC = step5\_tco.c $(SRC) $(LIBS\_SRC)

S6\_SRC = step6\_file.c $(SRC) $(LIBS\_SRC)

S7\_SRC = step7\_quote.c $(SRC) $(LIBS\_SRC)

S8\_SRC = step8\_macros.c $(SRC) $(LIBS\_SRC)

S9\_SRC = step9\_try.c $(SRC) $(LIBS\_SRC)

SA\_SRC = stepA\_mal.c $(SRC) $(LIBS\_SRC)

S0\_HEADERS =

S1\_HEADERS = reader.h types.h printer.h $(LIB\_LIST\_H)

S2\_HEADERS = reader.h types.h printer.h $(LIBS\_H)

S3\_HEADERS = reader.h types.h printer.h env.h $(LIBS\_H)

S4\_HEADERS = $(HEADERS) $(LIBS\_H)

S5\_HEADERS = $(HEADERS) $(LIBS\_H)

S6\_HEADERS = $(HEADERS) $(LIBS\_H)

S7\_HEADERS = $(HEADERS) $(LIBS\_H)

S8\_HEADERS = $(HEADERS) $(LIBS\_H)

S9\_HEADERS = $(HEADERS) $(LIBS\_H)

SA\_HEADERS = $(HEADERS) $(LIBS\_H)

S0 = step0\_repl

S1 = step1\_read\_print

S2 = step2\_eval

S3 = step3\_env

S4 = step4\_if\_fn\_do

S5 = step5\_tco

S6 = step6\_file

S7 = step7\_quote

S8 = step8\_macros

S9 = step9\_try

SA = stepA\_mal

all: $(S0) $(S1) $(S2) $(S3) $(S4) $(S5) $(S6) $(S7) $(S8) $(S9) $(SA)

$(S0): $(S0\_SRC) $(S0\_HEADERS)

$(CC) $(CFLAGS) $(S0\_SRC) $(LIBS) -o $(S0)

$(S1): $(S1\_SRC) $(S1\_HEADERS)

$(CC) $(CFLAGS) $(S1\_SRC) $(LIBS) -o $(S1)

$(S2): $(S2\_SRC) $(S2\_HEADERS)

$(CC) $(CFLAGS) $(S2\_SRC) $(LIBS) -o $(S2)

$(S3): $(S3\_SRC) $(S3\_HEADERS)

$(CC) $(CFLAGS) $(S3\_SRC) $(LIBS) -o $(S3)

$(S4): $(S4\_SRC) $(S4\_HEADERS)

$(CC) $(CFLAGS) $(S4\_SRC) $(LIBS) -o $(S4)

$(S5): $(S5\_SRC) $(S5\_HEADERS)

$(CC) $(CFLAGS) $(S5\_SRC) $(LIBS) -o $(S5)

$(S6): $(S6\_SRC) $(S6\_HEADERS)

$(CC) $(CFLAGS) $(S6\_SRC) $(LIBS) -o $(S6)

$(S7): $(S7\_SRC) $(S7\_HEADERS)

$(CC) $(CFLAGS) $(S7\_SRC) $(LIBS) -o $(S7)

$(S8): $(S8\_SRC) $(S8\_HEADERS)

$(CC) $(CFLAGS) $(S8\_SRC) $(LIBS) -o $(S8)

$(S9): $(S9\_SRC) $(S9\_HEADERS)

$(CC) $(CFLAGS) $(S9\_SRC) $(LIBS) -o $(S9)

$(SA): $(SA\_SRC) $(SA\_HEADERS)

$(CC) $(CFLAGS) $(SA\_SRC) $(LIBS) $(FFI\_LIBS) -DWITH\_FFI -o $(SA)

.PHONY clean:

rm -f $(S0) $(S1) $(S2) $(S3) $(S4) $(S5) $(S6) $(S7) $(S8) $(S9) $(SA)

***stepA\_mal.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#include "core.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define SYMBOL\_DO "do"

#define SYMBOL\_IF "if"

#define SYMBOL\_FNSTAR "fn\*"

#define SYMBOL\_QUOTE "quote"

#define SYMBOL\_QUASIQUOTE "quasiquote"

#define SYMBOL\_QUASIQUOTEEXPAND "quasiquoteexpand"

#define SYMBOL\_UNQUOTE "unquote"

#define SYMBOL\_SPLICE\_UNQUOTE "splice-unquote"

#define SYMBOL\_DEFMACROBANG "defmacro!"

#define SYMBOL\_MACROEXPAND "macroexpand"

#define SYMBOL\_TRYSTAR "try\*"

#define SYMBOL\_CATCHSTAR "catch\*"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\*\* env);

void eval\_letstar(MalType\*\* ast, Env\*\* env);

void eval\_if(MalType\*\* ast, Env\*\* env);

MalType\* eval\_fnstar(MalType\* ast, Env\* env);

MalType\* eval\_do(MalType\* ast, Env\* env);

MalType\* eval\_quote(MalType\* ast);

MalType\* eval\_quasiquote(MalType\* ast);

MalType\* eval\_quasiquoteexpand(MalType\* ast);

MalType\* eval\_defmacrobang(MalType\*, Env\*\* env);

MalType\* eval\_macroexpand(MalType\* ast, Env\* env);

MalType\* macroexpand(MalType\* ast, Env\* env);

void eval\_try(MalType\*\* ast, Env\*\* env);

/\* Use goto to jump here rather than calling eval for tail-call elimination \*/

TCE\_entry\_point:

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* macroexpansion \*/

ast = macroexpand(ast, env);

if (is\_error(ast)) { return ast; }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_letstar(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_IF) == 0) {

/\* TCE - modify ast directly and jump back to eval \*/

eval\_if(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_FNSTAR) == 0) {

return eval\_fnstar(ast, env);

}

else if (strcmp(symbol, SYMBOL\_DO) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

ast = eval\_do(ast, env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUOTE) == 0) {

return eval\_quote(ast);

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTE) == 0) {

ast = eval\_quasiquote(ast);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTEEXPAND) == 0) {

list lst = ast->value.mal\_list;

return eval\_quasiquote(make\_list(lst));

}

else if (strcmp(symbol, SYMBOL\_DEFMACROBANG) == 0) {

return eval\_defmacrobang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_MACROEXPAND) == 0) {

return eval\_macroexpand(ast, env);

}

else if (strcmp(symbol, SYMBOL\_TRYSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_try(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else if (is\_closure(func)) {

MalClosure\* closure = func->value.mal\_closure;

list params = (closure->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(evlst->next);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !closure->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

/\* TCE - modify ast and env directly and jump back to eval \*/

env = env\_make(closure->env, params, evlst->next, closure->more\_symbol);

ast = func->value.mal\_closure->definition;

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

else {

return make\_error\_fmt("first item in list is not callable: '%s'", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

/\* declare as global so it can be accessed by mal\_eval \*/

Env\* global\_env;

MalType\* mal\_eval(list args) {

MalType\* ast = args->data;

return EVAL(ast, global\_env);

}

MalType\* mal\_readline(list args) {

if (!args || args->next) {

return make\_error("'readline': expected exactly one argument");

}

MalType\* prompt = args->data;

if (!is\_string(prompt)) {

return make\_error\_fmt("'readline': argument is not a string '%s'", \

pr\_str(prompt, UNREADABLY));

}

char\* str = readline(prompt->value.mal\_string);

if (str) {

add\_history(str);

return make\_string(str);

}

else {

return make\_nil();

}

}

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

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

global\_env = repl\_env;

ns\* core = ns\_make\_core();

hashmap mappings = core->mappings;

while (mappings) {

char\* symbol = mappings->data;

MalType\*(\*function)(list) = mappings->next->data;

env\_set\_C\_fn(repl\_env, symbol, function);

/\* pop symbol and function from hashmap/list \*/

mappings = mappings->next->next;

}

env\_set\_C\_fn(repl\_env, "eval", mal\_eval);

env\_set\_C\_fn(repl\_env, "readline", mal\_readline);

/\* add functions written in mal - not using rep as it prints the result \*/

EVAL(READ("(def! not (fn\* (a) (if a false true)))"), repl\_env);

EVAL(READ("(def! load-file (fn\* (f) (eval (read-string (str \"(do \" (slurp f) \"\nnil)\")))))"), repl\_env);

EVAL(READ("(defmacro! cond (fn\* (& xs) (if (> (count xs) 0) (list 'if (first xs) (if (> (count xs) 1) (nth xs 1) (throw \"odd number of forms to cond\")) (cons 'cond (rest (rest xs)))))))"), repl\_env);

/\* make command line arguments available in the environment \*/

list lst = NULL;

for (long i = 2; i < argc; i++) {

lst = list\_push(lst, make\_string(argv[i]));

}

env\_set(repl\_env, make\_symbol("\*ARGV\*"), make\_list(list\_reverse(lst)));

env\_set(repl\_env, make\_symbol("\*host-language\*"), make\_string("c.2"));

/\* run in script mode if a filename is given \*/

if (argc > 1) {

/\* first argument on command line is filename \*/

char\* load\_command = snprintfbuf(1024, "(load-file \"%s\")", argv[1]);

EVAL(READ(load\_command), repl\_env);

}

/\* run in repl mode when no cmd line args \*/

else {

/\* Greeting message \*/

EVAL(READ("(println (str \"Mal [\" \*host-language\* \"]\"))"), repl\_env);

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'def!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'def!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)){

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

void eval\_letstar(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_error("'let\*': missing bindings list");

return;

}

MalType\* bindings = lst->next->data;

MalType\* forms = lst->next->next ? lst->next->next->data : make\_nil();

if (!is\_sequential(bindings)) {

\*ast = make\_error("'let\*': first argument is not list or vector");

return;

}

list bindings\_list = bindings->value.mal\_list;

if (list\_count(bindings\_list) % 2 == 1) {

\*ast = make\_error("'let\*': expected an even number of binding pairs");

return;

}

Env\* letstar\_env = env\_make(\*env, NULL, NULL, NULL);

/\* evaluate the bindings \*/

while(bindings\_list) {

MalType\* symbol = bindings\_list->data;

MalType\* value = EVAL(bindings\_list->next->data, letstar\_env);

/\* early return from error \*/

if (is\_error(value)) {

\*ast = value;

return;

}

env\_set(letstar\_env, symbol, value);

bindings\_list = bindings\_list->next->next;

}

\*env = letstar\_env;

\*ast = forms;

return;

}

void eval\_if(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next || !lst->next->next) {

\*ast = make\_error("'if': too few arguments");

return;

}

if (lst->next->next->next && lst->next->next->next->next) {

\*ast = make\_error("'if': too many arguments");

return;

}

MalType\* condition = EVAL(lst->next->data, \*env);

if (is\_error(condition)) {

\*ast = condition;

return;

}

if (is\_false(condition) || is\_nil(condition)) {

/\* check whether false branch is present \*/

if (lst->next->next->next) {

\*ast = lst->next->next->next->data;

return;

}

else {

\*ast = make\_nil();

return;

}

} else {

\*ast = lst->next->next->data;

return;

}

}

MalType\* eval\_fnstar(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* regularise\_parameters(list\* params, MalType\*\* more);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'fn\*': missing argument list");

}

else if (!lst->next->next) {

return make\_error("'fn\*': missing function body");

}

MalType\* params = lst->next->data;

list params\_list = params->value.mal\_list;

MalType\* more\_symbol = NULL;

MalType\* result = regularise\_parameters(&params\_list, &more\_symbol);

if (is\_error(result)) { return result; }

MalType\* definition = lst->next->next->data;

MalType\* regular\_params = make\_list(params\_list);

return make\_closure(env, regular\_params, definition, more\_symbol);

}

MalType\* eval\_do(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

/\* handle empty 'do' \*/

if (!lst->next) {

return make\_nil();

}

/\* evaluate all but the last form \*/

lst = lst->next;

while (lst->next) {

MalType\* val = EVAL(lst->data, env);

/\* return error early \*/

if (is\_error(val)) {

return val;

}

lst = lst->next;

}

/\* return the last form for TCE evaluation \*/

return lst->data;

}

MalType\* eval\_quote(MalType\* ast) {

list lst = (ast->value.mal\_list)->next;

if (!lst) {

return make\_nil();

}

else if (lst->next) {

return make\_error("'quote': expected exactly one argument");

}

else {

return lst->data;

}

}

MalType\* eval\_quasiquote(MalType\* ast) {

/\* forward reference \*/

MalType\* quasiquote(MalType\* ast);

list lst = ast->value.mal\_list;

/\* no arguments (quasiquote) \*/

if (!lst->next) {

return make\_nil();

}

/\* too many arguments \*/

else if (lst->next->next) {

return make\_error("'quasiquote': expected exactly one argument");

}

return quasiquote(lst->next->data);

}

MalType\* quasiquote(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

MalType\* quasiquote\_vector(MalType\* ast);

/\* argument to quasiquote is self-evaluating: (quasiquote val)

=> val \*/

if (is\_self\_evaluating(ast)) {

return ast;

}

/\* argument to quasiquote is a vector: (quasiquote [first rest]) \*/

else if (is\_vector(ast)) {

return quasiquote\_vector(ast);

}

/\* argument to quasiquote is a list: (quasiquote (first rest)) \*/

else if (is\_list(ast)){

return quasiquote\_list(ast);

}

/\* argument to quasiquote is not self-evaluating and isn't sequential: (quasiquote val)

=> (quote val) \*/

else {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

MalType\* quasiquote\_vector(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

list args = ast->value.mal\_list;

if (args) {

MalType\* first = args->data;

/\* if first element is unquote return quoted \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0) {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

/\* otherwise process like a list \*/

list lst = list\_make(make\_symbol("vec"));

MalType\* result = quasiquote\_list(ast);

if (is\_error(result)) {

return result;

} else {

lst = list\_push(lst, result);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

MalType\* quasiquote\_list(MalType\* ast) {

list args = ast->value.mal\_list;

/\* handle empty list: (quasiquote ())

=> () \*/

if (!args) {

return make\_list(NULL);

}

MalType\* first = args->data;

/\* handle unquote: (quasiquote (unquote second))

=> second \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0 && args->next) {

if (args->next->next) {

return make\_error("'quasiquote': unquote expected exactly one argument");

}

else {

return args->next->data;

}

}

/\* handle splice-unquote: (quasiquote ((splice-unquote first-second) rest))

=> (concat first-second (quasiquote rest)) \*/

else if (is\_list(first) &&

first->value.mal\_list != NULL &&

is\_symbol(first->value.mal\_list->data) &&

strcmp(((MalType\*)first->value.mal\_list->data)->value.mal\_symbol, SYMBOL\_SPLICE\_UNQUOTE) == 0) {

if (!first->value.mal\_list->next) {

return make\_error("'quasiquote': splice-unquote expected exactly one argument");

}

MalType\* first\_second = first->value.mal\_list->next->data;

list lst = list\_make(make\_symbol("concat"));

lst = list\_push(lst, first\_second);

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

}

lst = list\_push(lst, rest);

lst = list\_reverse(lst);

return make\_list(lst);

}

/\* handle all other lists recursively: (quasiquote (first rest))

=> (cons (quasiquote first) (quasiquote rest)) \*/

else {

list lst = list\_make(make\_symbol("cons"));

MalType\* first = quasiquote(args->data);

if (is\_error(first)) {

return first;

} else {

lst = list\_push(lst, first);

}

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

} else {

lst = list\_push(lst, rest);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

}

MalType\* eval\_defmacrobang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'defmacro!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'defmacro!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)) {

result = copy\_type(result);

result->is\_macro = 1;

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

MalType\* eval\_macroexpand(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* macroexpand(MalType\* ast, Env\* env);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_nil();

}

else if (lst->next->next) {

return make\_error("'macroexpand': expected exactly one argument");

}

else {

return macroexpand(lst->next->data, env);

}

}

MalType\* macroexpand(MalType\* ast, Env\* env) {

/\* forward reference \*/

int is\_macro\_call(MalType\* ast, Env\* env);

while(is\_macro\_call(ast, env)) {

list lst = ast->value.mal\_list;

MalType\* macro\_fn = env\_get(env, lst->data);

MalClosure\* cls = macro\_fn->value.mal\_closure;

MalType\* more\_symbol = cls->more\_symbol;

list params\_list = (cls->parameters)->value.mal\_list;

list args\_list = lst->next;

env = env\_make(cls->env, params\_list, args\_list, more\_symbol);

ast = EVAL(cls->definition, env);

}

return ast;

}

void eval\_try(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_nil();

return;

}

if (lst->next->next && lst->next->next->next) {

\*ast = make\_error("'try\*': expected maximum of two arguments");

return;

}

MalType\* try\_clause = lst->next->data;

MalType\* try\_result = EVAL(try\_clause, \*env);

/\* no catch\* clause \*/

if (!is\_error(try\_result) || !lst->next->next) {

\*ast = try\_result;

return;

}

/\* process catch\* clause \*/

MalType\* catch\_clause = lst->next->next->data;

list catch\_list = catch\_clause->value.mal\_list;

if (!catch\_list) {

\*ast = make\_error("'try\*': catch\* clause is empty");

return;

}

MalType\* catch\_symbol = catch\_list->data;

if (strcmp(catch\_symbol->value.mal\_symbol, SYMBOL\_CATCHSTAR) != 0) {

\*ast = make\_error("Error: catch clause is missing catch\* symbol");

return;

}

if (!catch\_list->next || !catch\_list->next->next) {

\*ast = make\_error("Error: catch\* clause expected two arguments");

return;

}

if (!is\_symbol(catch\_list->next->data)) {

\*ast = make\_error("Error: catch\* clause expected a symbol");

return;

}

/\* bind the symbol to the exception \*/

list symbol\_list = list\_make(catch\_list->next->data);

list expr\_list = list\_make(try\_result->value.mal\_error);

Env\* catch\_env = env\_make(\*env, symbol\_list, expr\_list, NULL);

\*ast = catch\_list->next->next->data;

\*env = catch\_env;

return;

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* regularise\_parameters(list\* args, MalType\*\* more\_symbol) {

/\* forward reference \*/

char\* symbol\_fn(gptr data);

list regular\_args = NULL;

while (\*args) {

MalType\* val = (\*args)->data;

if (!is\_symbol(val)) {

return make\_error\_fmt("non-symbol found in fn argument list '%s'", \

pr\_str(val, UNREADABLY));

}

if (val->value.mal\_symbol[0] == '&') {

/\* & is found but there is no symbol \*/

if (val->value.mal\_symbol[1] == '\0' && !(\*args)->next) {

return make\_error("missing symbol after '&' in argument list");

}

/\* & is found and there is a single symbol after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next &&

is\_symbol((\*args)->next->data) && !(\*args)->next->next)) {

\*more\_symbol = (\*args)->next->data;

break;

}

/\* & is found and there extra symbols after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next && (\*args)->next->next)) {

return make\_error\_fmt("unexpected symbol after '& %s' in argument list: '%s'", \

pr\_str((\*args)->next->data, UNREADABLY), \

pr\_str((\*args)->next->next->data, UNREADABLY));

}

/\* & is found as part of the symbol and no other symbols \*/

else if (val->value.mal\_symbol[1] != '\0' && !(\*args)->next) {

\*more\_symbol = make\_symbol((val->value.mal\_symbol + 1));

break;

}

/\* & is found as part of the symbol but there are other symbols after \*/

else if (val->value.mal\_symbol[1] != '\0' && (\*args)->next) {

return make\_error\_fmt("unexpected symbol after '%s' in argument list: '%s'", \

pr\_str(val, UNREADABLY), \

pr\_str((\*args)->next->data, UNREADABLY));

}

}

/\* & is not found - add the symbol to the regular argument list \*/

else {

if (list\_findf(regular\_args, val->value.mal\_symbol, symbol\_fn) > 0) {

return make\_error\_fmt("duplicate symbol in argument list: '%s'", \

pr\_str(val, UNREADABLY));

}

else {

regular\_args = list\_push(regular\_args, val);

}

}

\*args = (\*args)->next;

}

\*args = list\_reverse(regular\_args);

return make\_nil();

}

char\* symbol\_fn(gptr data) {

return (((MalType\*)data)->value.mal\_symbol);

}

/\* used by core functions but not EVAL as doesn't do TCE \*/

MalType\* apply(MalType\* fn, list args) {

if (is\_function(fn)) {

MalType\* (\*fun\_ptr)(list) = fn->value.mal\_function;

return (\*fun\_ptr)(args);

}

else { /\* is\_closure(fn) \*/

MalClosure\* c = fn->value.mal\_closure;

list params = (c->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(args);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !c->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

Env\* env = env\_make(c->env, params, args, c->more\_symbol);

return EVAL(fn->value.mal\_closure->definition, env);

}

}

}

int is\_macro\_call(MalType\* ast, Env\* env) {

/\* not a list \*/

if (!is\_list(ast)) {

return 0;

}

/\* empty list \*/

list lst = ast->value.mal\_list;

if (!lst) {

return 0;

}

/\* first item not a symbol \*/

MalType\* first = lst->data;

if (!is\_symbol(first)) {

return 0;

}

/\* lookup symbol \*/

MalType\* val = env\_get(env, first);

if (is\_error(val)) {

return 0;

}

else {

return (val->is\_macro);

}

}

***step6\_file.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#include "core.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define SYMBOL\_IF "if"

#define SYMBOL\_FNSTAR "fn\*"

#define SYMBOL\_DO "do"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\*\* env);

void eval\_letstar(MalType\*\* ast, Env\*\* env);

void eval\_if(MalType\*\* ast, Env\*\* env);

MalType\* eval\_fnstar(MalType\* ast, Env\* env);

MalType\* eval\_do(MalType\* ast, Env\* env);

/\* Use goto to jump here rather than calling eval for tail-call elimination \*/

TCE\_entry\_point:

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_letstar(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_DO) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

ast = eval\_do(ast, env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_IF) == 0) {

/\* TCE - modify ast directly and jump back to eval \*/

eval\_if(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_FNSTAR) == 0) {

return eval\_fnstar(ast, env);

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else if (is\_closure(func)) {

MalClosure\* closure = func->value.mal\_closure;

list params = (closure->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(evlst->next);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !closure->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

/\* TCE - modify ast and env directly and jump back to eval \*/

env = env\_make(closure->env, params, evlst->next, closure->more\_symbol);

ast = func->value.mal\_closure->definition;

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

else {

return make\_error\_fmt("first item in list is not callable: '%s'", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

/\* declare as global so it can be accessed by mal\_eval \*/

Env\* global\_env;

MalType\* mal\_eval(list args) {

MalType\* ast = args->data;

return EVAL(ast, global\_env);

}

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

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

global\_env = repl\_env;

ns\* core = ns\_make\_core();

hashmap mappings = core->mappings;

while (mappings) {

char\* symbol = mappings->data;

MalType\*(\*function)(list) = (MalType\*(\*)(list))mappings->next->data;

env\_set\_C\_fn(repl\_env, symbol, function);

/\* pop symbol and function from hashmap/list \*/

mappings = mappings->next->next;

}

env\_set\_C\_fn(repl\_env, "eval", mal\_eval);

/\* add functions written in mal - not using rep as it prints the result \*/

EVAL(READ("(def! not (fn\* (a) (if a false true)))"), repl\_env);

EVAL(READ("(def! load-file (fn\* (f) (eval (read-string (str \"(do \" (slurp f) \"\nnil)\")))))"), repl\_env);

/\* make command line arguments available in the environment \*/

list lst = NULL;

for (int i = 2; i < argc; i++) {

lst = list\_push(lst, make\_string(argv[i]));

}

env\_set(repl\_env, make\_symbol("\*ARGV\*"), make\_list(list\_reverse(lst)));

/\* run in script mode if a filename is given \*/

if (argc > 1) {

/\* first argument on command line is filename \*/

char\* load\_command = snprintfbuf(1024, "(load-file \"%s\")", argv[1]);

EVAL(READ(load\_command), repl\_env);

}

/\* run in repl mode when no cmd line args \*/

else {

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.6\n");

puts("Press Ctrl+d to exit\n");

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'def!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'def!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)){

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

void eval\_letstar(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_error("'let\*': missing bindings list");

return;

}

MalType\* bindings = lst->next->data;

MalType\* forms = lst->next->next ? lst->next->next->data : make\_nil();

if (!is\_sequential(bindings)) {

\*ast = make\_error("'let\*': first argument is not list or vector");

return;

}

list bindings\_list = bindings->value.mal\_list;

if (list\_count(bindings\_list) % 2 == 1) {

\*ast = make\_error("'let\*': expected an even number of binding pairs");

return;

}

Env\* letstar\_env = env\_make(\*env, NULL, NULL, NULL);

/\* evaluate the bindings \*/

while(bindings\_list) {

MalType\* symbol = bindings\_list->data;

MalType\* value = EVAL(bindings\_list->next->data, letstar\_env);

/\* early return from error \*/

if (is\_error(value)) {

\*ast = value;

return;

}

env\_set(letstar\_env, symbol, value);

bindings\_list = bindings\_list->next->next;

}

\*env = letstar\_env;

\*ast = forms;

return;

}

void eval\_if(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next || !lst->next->next) {

\*ast = make\_error("'if': too few arguments");

return;

}

if (lst->next->next->next && lst->next->next->next->next) {

\*ast = make\_error("'if': too many arguments");

return;

}

MalType\* condition = EVAL(lst->next->data, \*env);

if (is\_error(condition)) {

\*ast = condition;

return;

}

if (is\_false(condition) || is\_nil(condition)) {

/\* check whether false branch is present \*/

if (lst->next->next->next) {

\*ast = lst->next->next->next->data;

return;

}

else {

\*ast = make\_nil();

return;

}

} else {

\*ast = lst->next->next->data;

return;

}

}

MalType\* eval\_fnstar(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* regularise\_parameters(list\* params, MalType\*\* more);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'fn\*': missing argument list");

}

else if (!lst->next->next) {

return make\_error("'fn\*': missing function body");

}

MalType\* params = lst->next->data;

list params\_list = params->value.mal\_list;

MalType\* more\_symbol = NULL;

MalType\* result = regularise\_parameters(&params\_list, &more\_symbol);

if (is\_error(result)) { return result; }

MalType\* definition = lst->next->next->data;

MalType\* regular\_params = make\_list(params\_list);

return make\_closure(env, regular\_params, definition, more\_symbol);

}

MalType\* eval\_do(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

/\* handle empty 'do' \*/

if (!lst->next) {

return make\_nil();

}

/\* evaluate all but the last form \*/

lst = lst->next;

while (lst->next) {

MalType\* val = EVAL(lst->data, env);

/\* return error early \*/

if (is\_error(val)) {

return val;

}

lst = lst->next;

}

/\* return the last form for TCE evaluation \*/

return lst->data;

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* regularise\_parameters(list\* args, MalType\*\* more\_symbol) {

/\* forward reference \*/

char\* symbol\_fn(gptr data);

list regular\_args = NULL;

while (\*args) {

MalType\* val = (\*args)->data;

if (!is\_symbol(val)) {

return make\_error\_fmt("non-symbol found in fn argument list '%s'", \

pr\_str(val, UNREADABLY));

}

if (val->value.mal\_symbol[0] == '&') {

/\* & is found but there is no symbol \*/

if (val->value.mal\_symbol[1] == '\0' && !(\*args)->next) {

return make\_error("missing symbol after '&' in argument list");

}

/\* & is found and there is a single symbol after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next &&

is\_symbol((\*args)->next->data) && !(\*args)->next->next)) {

/\* TODO: check symbol is no a duplicate of one already on the list \*/

\*more\_symbol = (\*args)->next->data;

break;

}

/\* & is found and there extra symbols after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next && (\*args)->next->next)) {

return make\_error\_fmt("unexpected symbol after '& %s' in argument list: '%s'", \

pr\_str((\*args)->next->data, UNREADABLY), \

pr\_str((\*args)->next->next->data, UNREADABLY));

}

/\* & is found as part of the symbol and no other symbols \*/

else if (val->value.mal\_symbol[1] != '\0' && !(\*args)->next) {

\*more\_symbol = make\_symbol((val->value.mal\_symbol + 1));

break;

}

/\* & is found as part of the symbol but there are other symbols after \*/

else if (val->value.mal\_symbol[1] != '\0' && (\*args)->next) {

return make\_error\_fmt("unexpected symbol after '%s' in argument list: '%s'", \

pr\_str(val, UNREADABLY), \

pr\_str((\*args)->next->data, UNREADABLY));

}

}

/\* & is not found - add the symbol to the regular argument list \*/

else {

if (list\_findf(regular\_args, val->value.mal\_symbol, symbol\_fn) > 0) {

return make\_error\_fmt("duplicate symbol in argument list: '%s'", pr\_str(val, UNREADABLY));

}

else {

regular\_args = list\_push(regular\_args, val);

}

}

\*args = (\*args)->next;

}

\*args = list\_reverse(regular\_args);

return make\_nil();

}

char\* symbol\_fn(gptr data) {

return (((MalType\*)data)->value.mal\_symbol);

}

/\* used by core functions but not EVAL as doesn't do TCE \*/

MalType\* apply(MalType\* fn, list args) {

if (is\_function(fn)) {

MalType\* (\*fun\_ptr)(list) = fn->value.mal\_function;

return (\*fun\_ptr)(args);

}

else { /\* is\_closure(fn) \*/

MalClosure\* c = fn->value.mal\_closure;

list params = (c->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(args);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !c->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

Env\* env = env\_make(c->env, params, args, c->more\_symbol);

return EVAL(fn->value.mal\_closure->definition, env);

}

}

}

***step7\_quote.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#include "core.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define SYMBOL\_DO "do"

#define SYMBOL\_IF "if"

#define SYMBOL\_FNSTAR "fn\*"

#define SYMBOL\_QUOTE "quote"

#define SYMBOL\_QUASIQUOTE "quasiquote"

#define SYMBOL\_QUASIQUOTEEXPAND "quasiquoteexpand"

#define SYMBOL\_UNQUOTE "unquote"

#define SYMBOL\_SPLICE\_UNQUOTE "splice-unquote"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\*\* env);

void eval\_letstar(MalType\*\* ast, Env\*\* env);

void eval\_if(MalType\*\* ast, Env\*\* env);

MalType\* eval\_fnstar(MalType\* ast, Env\* env);

MalType\* eval\_do(MalType\* ast, Env\* env);

MalType\* eval\_quote(MalType\* ast);

MalType\* eval\_quasiquote(MalType\* ast);

MalType\* eval\_quasiquoteexpand(MalType\* ast);

/\* Use goto to jump here rather than calling eval for tail-call elimination \*/

TCE\_entry\_point:

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_letstar(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_IF) == 0) {

/\* TCE - modify ast directly and jump back to eval \*/

eval\_if(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_FNSTAR) == 0) {

return eval\_fnstar(ast, env);

}

else if (strcmp(symbol, SYMBOL\_DO) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

ast = eval\_do(ast, env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUOTE) == 0) {

return eval\_quote(ast);

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTE) == 0) {

ast = eval\_quasiquote(ast);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTEEXPAND) == 0) {

list lst = ast->value.mal\_list;

return eval\_quasiquote(make\_list(lst));

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else if (is\_closure(func)) {

MalClosure\* closure = func->value.mal\_closure;

list params = (closure->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(evlst->next);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !closure->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

/\* TCE - modify ast and env directly and jump back to eval \*/

env = env\_make(closure->env, params, evlst->next, closure->more\_symbol);

ast = func->value.mal\_closure->definition;

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

else {

return make\_error\_fmt("first item in list is not callable: '%s'", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

/\* declare as global so it can be accessed by mal\_eval \*/

Env\* global\_env;

MalType\* mal\_eval(list args) {

MalType\* ast = args->data;

return EVAL(ast, global\_env);

}

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

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

global\_env = repl\_env;

ns\* core = ns\_make\_core();

hashmap mappings = core->mappings;

while (mappings) {

char\* symbol = mappings->data;

MalType\*(\*function)(list) = (MalType\*(\*)(list))mappings->next->data;

env\_set\_C\_fn(repl\_env, symbol, function);

/\* pop symbol and function from hashmap/list \*/

mappings = mappings->next->next;

}

env\_set\_C\_fn(repl\_env, "eval", mal\_eval);

/\* add functions written in mal - not using rep as it prints the result \*/

EVAL(READ("(def! not (fn\* (a) (if a false true)))"), repl\_env);

EVAL(READ("(def! load-file (fn\* (f) (eval (read-string (str \"(do \" (slurp f) \"\nnil)\")))))"), repl\_env);

/\* make command line arguments available in the environment \*/

list lst = NULL;

for (long i = 2; i < argc; i++) {

lst = list\_push(lst, make\_string(argv[i]));

}

env\_set(repl\_env, make\_symbol("\*ARGV\*"), make\_list(list\_reverse(lst)));

/\* run in script mode if a filename is given \*/

if (argc > 1) {

/\* first argument on command line is filename \*/

char\* load\_command = snprintfbuf(1024, "(load-file \"%s\")", argv[1]);

EVAL(READ(load\_command), repl\_env);

}

/\* run in repl mode when no cmd line args \*/

else {

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.7\n");

puts("Press Ctrl+d to exit\n");

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'def!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'def!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)){

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

void eval\_letstar(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_error("'let\*': missing bindings list");

return;

}

MalType\* bindings = lst->next->data;

MalType\* forms = lst->next->next ? lst->next->next->data : make\_nil();

if (!is\_sequential(bindings)) {

\*ast = make\_error("'let\*': first argument is not list or vector");

return;

}

list bindings\_list = bindings->value.mal\_list;

if (list\_count(bindings\_list) % 2 == 1) {

\*ast = make\_error("'let\*': expected an even number of binding pairs");

return;

}

Env\* letstar\_env = env\_make(\*env, NULL, NULL, NULL);

/\* evaluate the bindings \*/

while(bindings\_list) {

MalType\* symbol = bindings\_list->data;

MalType\* value = EVAL(bindings\_list->next->data, letstar\_env);

/\* early return from error \*/

if (is\_error(value)) {

\*ast = value;

return;

}

env\_set(letstar\_env, symbol, value);

bindings\_list = bindings\_list->next->next;

}

\*env = letstar\_env;

\*ast = forms;

return;

}

void eval\_if(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next || !lst->next->next) {

\*ast = make\_error("'if': too few arguments");

return;

}

if (lst->next->next->next && lst->next->next->next->next) {

\*ast = make\_error("'if': too many arguments");

return;

}

MalType\* condition = EVAL(lst->next->data, \*env);

if (is\_error(condition)) {

\*ast = condition;

return;

}

if (is\_false(condition) || is\_nil(condition)) {

/\* check whether false branch is present \*/

if (lst->next->next->next) {

\*ast = lst->next->next->next->data;

return;

}

else {

\*ast = make\_nil();

return;

}

} else {

\*ast = lst->next->next->data;

return;

}

}

MalType\* eval\_fnstar(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* regularise\_parameters(list\* params, MalType\*\* more);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'fn\*': missing argument list");

}

else if (!lst->next->next) {

return make\_error("'fn\*': missing function body");

}

MalType\* params = lst->next->data;

list params\_list = params->value.mal\_list;

MalType\* more\_symbol = NULL;

MalType\* result = regularise\_parameters(&params\_list, &more\_symbol);

if (is\_error(result)) { return result; }

MalType\* definition = lst->next->next->data;

MalType\* regular\_params = make\_list(params\_list);

return make\_closure(env, regular\_params, definition, more\_symbol);

}

MalType\* eval\_do(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

/\* handle empty 'do' \*/

if (!lst->next) {

return make\_nil();

}

/\* evaluate all but the last form \*/

lst = lst->next;

while (lst->next) {

MalType\* val = EVAL(lst->data, env);

/\* return error early \*/

if (is\_error(val)) {

return val;

}

lst = lst->next;

}

/\* return the last form for TCE evaluation \*/

return lst->data;

}

MalType\* eval\_quote(MalType\* ast) {

list lst = (ast->value.mal\_list)->next;

if (!lst) {

return make\_nil();

}

else if (lst->next) {

return make\_error("'quote': expected exactly one argument");

}

else {

return lst->data;

}

}

MalType\* eval\_quasiquote(MalType\* ast) {

/\* forward reference \*/

MalType\* quasiquote(MalType\* ast);

list lst = ast->value.mal\_list;

/\* no arguments (quasiquote) \*/

if (!lst->next) {

return make\_nil();

}

/\* too many arguments \*/

else if (lst->next->next) {

return make\_error("'quasiquote': expected exactly one argument");

}

return quasiquote(lst->next->data);

}

MalType\* quasiquote(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

MalType\* quasiquote\_vector(MalType\* ast);

/\* argument to quasiquote is self-evaluating: (quasiquote val)

=> val \*/

if (is\_self\_evaluating(ast)) {

return ast;

}

/\* argument to quasiquote is a vector: (quasiquote [first rest]) \*/

else if (is\_vector(ast)) {

return quasiquote\_vector(ast);

}

/\* argument to quasiquote is a list: (quasiquote (first rest)) \*/

else if (is\_list(ast)){

return quasiquote\_list(ast);

}

/\* argument to quasiquote is not self-evaluating and isn't sequential: (quasiquote val)

=> (quote val) \*/

else {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

MalType\* quasiquote\_vector(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

list args = ast->value.mal\_list;

if (args) {

MalType\* first = args->data;

/\* if first element is unquote return quoted \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0) {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

/\* otherwise process like a list \*/

list lst = list\_make(make\_symbol("vec"));

MalType\* result = quasiquote\_list(ast);

if (is\_error(result)) {

return result;

} else {

lst = list\_push(lst, result);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

MalType\* quasiquote\_list(MalType\* ast) {

list args = ast->value.mal\_list;

/\* handle empty list: (quasiquote ())

=> () \*/

if (!args) {

return make\_list(NULL);

}

MalType\* first = args->data;

/\* handle unquote: (quasiquote (unquote second))

=> second \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0 && args->next) {

if (args->next->next) {

return make\_error("'quasiquote': unquote expected exactly one argument");

}

else {

return args->next->data;

}

}

/\* handle splice-unquote: (quasiquote ((splice-unquote first-second) rest))

=> (concat first-second (quasiquote rest)) \*/

else if (is\_list(first) &&

first->value.mal\_list != NULL &&

is\_symbol(first->value.mal\_list->data) &&

strcmp(((MalType\*)first->value.mal\_list->data)->value.mal\_symbol, SYMBOL\_SPLICE\_UNQUOTE) == 0) {

if (!first->value.mal\_list->next) {

return make\_error("'quasiquote': splice-unquote expected exactly one argument");

}

MalType\* first\_second = first->value.mal\_list->next->data;

list lst = list\_make(make\_symbol("concat"));

lst = list\_push(lst, first\_second);

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

}

lst = list\_push(lst, rest);

lst = list\_reverse(lst);

return make\_list(lst);

}

/\* handle all other lists recursively: (quasiquote (first rest))

=> (cons (quasiquote first) (quasiquote rest)) \*/

else {

list lst = list\_make(make\_symbol("cons"));

MalType\* first = quasiquote(args->data);

if (is\_error(first)) {

return first;

} else {

lst = list\_push(lst, first);

}

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

} else {

lst = list\_push(lst, rest);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* regularise\_parameters(list\* args, MalType\*\* more\_symbol) {

/\* forward reference \*/

char\* symbol\_fn(gptr data);

list regular\_args = NULL;

while (\*args) {

MalType\* val = (\*args)->data;

if (!is\_symbol(val)) {

return make\_error\_fmt("non-symbol found in fn argument list '%s'", \

pr\_str(val, UNREADABLY));

}

if (val->value.mal\_symbol[0] == '&') {

/\* & is found but there is no symbol \*/

if (val->value.mal\_symbol[1] == '\0' && !(\*args)->next) {

return make\_error("missing symbol after '&' in argument list");

}

/\* & is found and there is a single symbol after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next &&

is\_symbol((\*args)->next->data) && !(\*args)->next->next)) {

/\* TODO: check symbol is no a duplicate of one already on the list \*/

\*more\_symbol = (\*args)->next->data;

break;

}

/\* & is found and there extra symbols after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next && (\*args)->next->next)) {

return make\_error\_fmt("unexpected symbol after '& %s' in argument list: '%s'", \

pr\_str((\*args)->next->data, UNREADABLY), \

pr\_str((\*args)->next->next->data, UNREADABLY));

}

/\* & is found as part of the symbol and no other symbols \*/

else if (val->value.mal\_symbol[1] != '\0' && !(\*args)->next) {

\*more\_symbol = make\_symbol((val->value.mal\_symbol + 1));

break;

}

/\* & is found as part of the symbol but there are other symbols after \*/

else if (val->value.mal\_symbol[1] != '\0' && (\*args)->next) {

return make\_error\_fmt("unexpected symbol after '%s' in argument list: '%s'", \

pr\_str(val, UNREADABLY), \

pr\_str((\*args)->next->data, UNREADABLY));

}

}

/\* & is not found - add the symbol to the regular argument list \*/

else {

if (list\_findf(regular\_args, val->value.mal\_symbol, symbol\_fn) > 0) {

return make\_error\_fmt("duplicate symbol in argument list: '%s'", pr\_str(val, UNREADABLY));

}

else {

regular\_args = list\_push(regular\_args, val);

}

}

\*args = (\*args)->next;

}

\*args = list\_reverse(regular\_args);

return make\_nil();

}

char\* symbol\_fn(gptr data) {

return (((MalType\*)data)->value.mal\_symbol);

}

/\* used by core functions but not EVAL as doesn't do TCE \*/

MalType\* apply(MalType\* fn, list args) {

if (is\_function(fn)) {

MalType\* (\*fun\_ptr)(list) = fn->value.mal\_function;

return (\*fun\_ptr)(args);

}

else { /\* is\_closure(fn) \*/

MalClosure\* c = fn->value.mal\_closure;

list params = (c->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(args);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !c->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

Env\* env = env\_make(c->env, params, args, c->more\_symbol);

return EVAL(fn->value.mal\_closure->definition, env);

}

}

}

***core.c***

#include <stdio.h>

#include <string.h>

#include <sys/time.h>

#include <math.h>

#include <gc.h>

/\* only needed for ffi \*/

#ifdef WITH\_FFI

#include <dlfcn.h>

#include <ffi.h>

#endif

#include "libs/hashmap/hashmap.h"

#include "core.h"

#include "types.h"

#include "printer.h"

#include "reader.h"

#include "env.h"

#define STRING\_BUFFER\_SIZE 128

/\* forward references to main file \*/

MalType\* apply(MalType\* fn, list args);

/\* core ns functions \*/

MalType\* mal\_add(list);

MalType\* mal\_sub(list);

MalType\* mal\_mul(list);

MalType\* mal\_div(list);

MalType\* mal\_prn(list);

MalType\* mal\_println(list);

MalType\* mal\_pr\_str(list);

MalType\* mal\_str(list);

MalType\* mal\_read\_string(list);

MalType\* mal\_slurp(list);

MalType\* mal\_list(list);

MalType\* mal\_list\_questionmark(list);

MalType\* mal\_empty\_questionmark(list);

MalType\* mal\_count(list);

MalType\* mal\_cons(list);

MalType\* mal\_concat(list);

MalType\* mal\_nth(list);

MalType\* mal\_first(list);

MalType\* mal\_rest(list);

MalType\* mal\_equals(list);

MalType\* mal\_lessthan(list);

MalType\* mal\_lessthanorequalto(list);

MalType\* mal\_greaterthan(list);

MalType\* mal\_greaterthanorequalto(list);

MalType\* mal\_atom(list);

MalType\* mal\_atom\_questionmark(list);

MalType\* mal\_deref(list);

MalType\* mal\_reset\_bang(list);

MalType\* mal\_swap\_bang(list);

MalType\* mal\_throw(list);

MalType\* mal\_apply(list);

MalType\* mal\_map(list);

MalType\* mal\_nil\_questionmark(list);

MalType\* mal\_true\_questionmark(list);

MalType\* mal\_false\_questionmark(list);

MalType\* mal\_symbol\_questionmark(list);

MalType\* mal\_keyword\_questionmark(list);

MalType\* mal\_symbol(list);

MalType\* mal\_keyword(list);

MalType\* mal\_vec(list);

MalType\* mal\_vector(list);

MalType\* mal\_vector\_questionmark(list);

MalType\* mal\_sequential\_questionmark(list);

MalType\* mal\_hash\_map(list);

MalType\* mal\_map\_questionmark(list);

MalType\* mal\_assoc(list);

MalType\* mal\_dissoc(list);

MalType\* mal\_get(list);

MalType\* mal\_contains\_questionmark(list);

MalType\* mal\_keys(list);

MalType\* mal\_vals(list);

MalType\* mal\_string\_questionmark(list);

MalType\* mal\_number\_questionmark(list);

MalType\* mal\_fn\_questionmark(list);

MalType\* mal\_macro\_questionmark(list);

MalType\* mal\_time\_ms(list);

MalType\* mal\_conj(list);

MalType\* mal\_seq(list);

MalType\* mal\_meta(list);

MalType\* mal\_with\_meta(list);

/\* only needed for ffi \*/

#ifdef WITH\_FFI

MalType\* mal\_dot(list);

#endif

ns\* ns\_make\_core() {

ns\* core = GC\_MALLOC(sizeof(\*core));

hashmap core\_functions = NULL;

/\* arithmetic \*/

core\_functions = hashmap\_put(core\_functions, "+", mal\_add);

core\_functions = hashmap\_put(core\_functions, "-", mal\_sub);

core\_functions = hashmap\_put(core\_functions, "\*", mal\_mul);

core\_functions = hashmap\_put(core\_functions, "/", mal\_div);

/\* strings \*/

core\_functions = hashmap\_put(core\_functions, "prn", mal\_prn);

core\_functions = hashmap\_put(core\_functions, "pr-str", mal\_pr\_str);

core\_functions = hashmap\_put(core\_functions, "str", mal\_str);

core\_functions = hashmap\_put(core\_functions, "println", mal\_println);

core\_functions = hashmap\_put(core\_functions, "read-string", mal\_read\_string);

/\* files \*/

core\_functions = hashmap\_put(core\_functions, "slurp", mal\_slurp);

/\* lists \*/

core\_functions = hashmap\_put(core\_functions, "list", mal\_list);

core\_functions = hashmap\_put(core\_functions, "empty?", mal\_empty\_questionmark);

core\_functions = hashmap\_put(core\_functions, "count", mal\_count);

core\_functions = hashmap\_put(core\_functions, "cons", mal\_cons);

core\_functions = hashmap\_put(core\_functions, "concat", mal\_concat);

core\_functions = hashmap\_put(core\_functions, "nth", mal\_nth);

core\_functions = hashmap\_put(core\_functions, "first", mal\_first);

core\_functions = hashmap\_put(core\_functions, "rest", mal\_rest);

/\* predicates \*/

core\_functions = hashmap\_put(core\_functions, "=", mal\_equals);

core\_functions = hashmap\_put(core\_functions, "<", mal\_lessthan);

core\_functions = hashmap\_put(core\_functions, "<=", mal\_lessthanorequalto);

core\_functions = hashmap\_put(core\_functions, ">", mal\_greaterthan);

core\_functions = hashmap\_put(core\_functions, ">=", mal\_greaterthanorequalto);

core\_functions = hashmap\_put(core\_functions, "list?", mal\_list\_questionmark);

core\_functions = hashmap\_put(core\_functions, "nil?", mal\_nil\_questionmark);

core\_functions = hashmap\_put(core\_functions, "true?", mal\_true\_questionmark);

core\_functions = hashmap\_put(core\_functions, "false?", mal\_false\_questionmark);

core\_functions = hashmap\_put(core\_functions, "symbol?", mal\_symbol\_questionmark);

core\_functions = hashmap\_put(core\_functions, "keyword?", mal\_keyword\_questionmark);

core\_functions = hashmap\_put(core\_functions, "vector?", mal\_vector\_questionmark);

core\_functions = hashmap\_put(core\_functions, "sequential?", mal\_sequential\_questionmark);

core\_functions = hashmap\_put(core\_functions, "map?", mal\_map\_questionmark);

core\_functions = hashmap\_put(core\_functions, "string?", mal\_string\_questionmark);

core\_functions = hashmap\_put(core\_functions, "number?", mal\_number\_questionmark);

core\_functions = hashmap\_put(core\_functions, "fn?", mal\_fn\_questionmark);

core\_functions = hashmap\_put(core\_functions, "macro?", mal\_macro\_questionmark);

/\* atoms \*/

core\_functions = hashmap\_put(core\_functions, "atom", mal\_atom);

core\_functions = hashmap\_put(core\_functions, "atom?", mal\_atom\_questionmark);

core\_functions = hashmap\_put(core\_functions, "deref", mal\_deref);

core\_functions = hashmap\_put(core\_functions, "reset!", mal\_reset\_bang);

core\_functions = hashmap\_put(core\_functions, "swap!", mal\_swap\_bang);

/\* other \*/

core\_functions = hashmap\_put(core\_functions, "throw", mal\_throw);

core\_functions = hashmap\_put(core\_functions, "apply", mal\_apply);

core\_functions = hashmap\_put(core\_functions, "map", mal\_map);

core\_functions = hashmap\_put(core\_functions, "symbol", mal\_symbol);

core\_functions = hashmap\_put(core\_functions, "keyword", mal\_keyword);

core\_functions = hashmap\_put(core\_functions, "vec", mal\_vec);

core\_functions = hashmap\_put(core\_functions, "vector", mal\_vector);

core\_functions = hashmap\_put(core\_functions, "hash-map", mal\_hash\_map);

/\* hash-maps \*/

core\_functions = hashmap\_put(core\_functions, "contains?", mal\_contains\_questionmark);

core\_functions = hashmap\_put(core\_functions, "assoc", mal\_assoc);

core\_functions = hashmap\_put(core\_functions, "dissoc", mal\_dissoc);

core\_functions = hashmap\_put(core\_functions, "get", mal\_get);

core\_functions = hashmap\_put(core\_functions, "keys", mal\_keys);

core\_functions = hashmap\_put(core\_functions, "vals", mal\_vals);

/\* misc \*/

core\_functions = hashmap\_put(core\_functions, "time-ms", mal\_time\_ms);

core\_functions = hashmap\_put(core\_functions, "conj", mal\_conj);

core\_functions = hashmap\_put(core\_functions, "seq", mal\_seq);

core\_functions = hashmap\_put(core\_functions, "meta", mal\_meta);

core\_functions = hashmap\_put(core\_functions, "with-meta", mal\_with\_meta);

/\* only needed for ffi \*/

#ifdef WITH\_FFI

core\_functions = hashmap\_put(core\_functions, ".", mal\_dot);

#endif

core->mappings = core\_functions;

return core;

}

/\* core function definitons \*/

MalType\* mal\_add(list args) {

/\* Accepts any number of arguments \*/

int return\_float = 0;

long i\_sum = 0;

double r\_sum = 0.0;

while(args) {

MalType\* val = args->data;

if (!is\_number(val)) {

return make\_error("'+': expected numerical arguments");

}

if (is\_integer(val) && !return\_float) {

i\_sum = i\_sum + val->value.mal\_integer;

}

else if (is\_integer(val)) {

r\_sum = (double)i\_sum + r\_sum + val->value.mal\_integer;

i\_sum = 0;

}

else {

r\_sum = (double)i\_sum + r\_sum + val->value.mal\_float;

i\_sum = 0;

return\_float = 1;

}

args = args->next;

}

if (return\_float) {

return make\_float(r\_sum);

} else {

return make\_integer(i\_sum);

}

}

MalType\* mal\_sub(list args) {

/\* Accepts any number of arguments \*/

int return\_float = 0;

long i\_sum = 0;

double r\_sum = 0.0;

if (args) {

MalType\* val = args->data;

args = args->next;

if (!is\_number(val)) {

return make\_error\_fmt("'-': expected numerical arguments");

}

if (is\_integer(val)) {

i\_sum = val->value.mal\_integer;

} else {

r\_sum = val->value.mal\_float;

return\_float = 1;

}

while(args) {

val = args->data;

if (!is\_number(val)) {

return make\_error\_fmt("'-': expected numerical arguments");

}

if (is\_integer(val) && !return\_float) {

i\_sum = i\_sum - val->value.mal\_integer;

}

else if (is\_integer(val)) {

r\_sum = (double)i\_sum + r\_sum - (double)val->value.mal\_integer;

i\_sum = 0;

}

else {

r\_sum = (double)i\_sum + r\_sum - val->value.mal\_float;

i\_sum = 0;

return\_float = 1;

}

args = args->next;

}

}

if (return\_float) {

return make\_float(r\_sum);

} else {

return make\_integer(i\_sum);

}

}

MalType\* mal\_mul(list args) {

/\* Accepts any number of arguments \*/

int return\_float = 0;

long i\_product = 1;

double r\_product = 1.0;

while(args) {

MalType\* val = args->data;

if (!is\_number(val)) {

return make\_error\_fmt("'\*': expected numerical arguments");

}

if (is\_integer(val) && !return\_float) {

i\_product \*= val->value.mal\_integer;

}

else if (is\_integer(val)) {

r\_product \*= (double)val->value.mal\_integer;

r\_product \*= (double)i\_product;

i\_product = 1;

}

else {

r\_product \*= (double)i\_product;

r\_product \*= val->value.mal\_float;

i\_product = 1;

return\_float = 1;

}

args = args->next;

}

if (return\_float) {

return make\_float(r\_product);

} else {

return make\_integer(i\_product);

}

}

MalType\* mal\_div(list args) {

/\* Accepts any number of arguments \*/

int return\_float = 0;

long i\_product = 1;

double r\_product = 1.0;

if (args) {

MalType\* val = args->data;

if (!is\_number(val)) {

return make\_error\_fmt("'/': expected numerical arguments");

}

if (is\_integer(val)) {

i\_product = val->value.mal\_integer;

} else {

r\_product = val->value.mal\_float;

return\_float = 1;

}

args = args->next;

while(args) {

val = args->data;

if (!is\_number(val)) {

return make\_error\_fmt("'/': expected numerical arguments");

}

/\* integer division \*/

if (is\_integer(val) && !return\_float) {

i\_product /= val->value.mal\_integer;

}

/\* promote integer to double \*/

else if (is\_integer(val)) {

if (i\_product != 1) {

r\_product = (double)i\_product / (double)val->value.mal\_integer;

i\_product = 1;

} else {

r\_product /= (double)val->value.mal\_integer;

}

}

/\* double division \*/

else {

return\_float = 1;

if (i\_product != 1) {

r\_product = (double)i\_product / val->value.mal\_float;

i\_product = 1;

} else {

r\_product /= val->value.mal\_float;

}

}

args = args->next;

}

}

if (return\_float) {

return make\_float(r\_product);

} else {

return make\_integer(i\_product);

}

}

MalType\* mal\_lessthan(list args) {

if (!args || !args->next || args->next->next) {

return make\_error\_fmt("'<': expected exactly two arguments");

}

MalType\* first\_val = args->data;

MalType\* second\_val = args->next->data;

if (!is\_number(first\_val) || !is\_number(second\_val)) {

return make\_error\_fmt("'<': expected numerical arguments");

}

int cmp = 0;

if (is\_integer(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_integer < second\_val->value.mal\_integer);

}

else if (is\_integer(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_integer < second\_val->value.mal\_float);

}

else if (is\_float(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_float < second\_val->value.mal\_integer);

}

else if (is\_float(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_float < second\_val->value.mal\_float);

}

else {

/\* shouldn't happen unless new numerical type is added \*/

return make\_error\_fmt("'<': unknown numerical type");

}

if (cmp) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_lessthanorequalto(list args) {

if (!args || !args->next || args->next->next) {

return make\_error\_fmt("'<=': expected exactly two arguments");

}

MalType\* first\_val = args->data;

MalType\* second\_val = args->next->data;

if (!is\_number(first\_val) || !is\_number(second\_val)) {

return make\_error\_fmt("'<=': expected numerical arguments");

}

int cmp = 0;

if (is\_integer(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_integer <= second\_val->value.mal\_integer);

}

else if (is\_integer(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_integer <= second\_val->value.mal\_float);

}

else if (is\_float(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_float <= second\_val->value.mal\_integer);

}

else if (is\_float(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_float < second\_val->value.mal\_float);

}

else {

/\* shouldn't happen unless new numerical type is added \*/

return make\_error\_fmt("'<=': unknown numerical type");

}

if (cmp) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_greaterthan(list args) {

if (!args || !args->next || args->next->next) {

return make\_error\_fmt("'>': expected exactly two arguments");

}

MalType\* first\_val = args->data;

MalType\* second\_val = args->next->data;

if (!is\_number(first\_val) || !is\_number(second\_val)) {

return make\_error\_fmt("'>': expected numerical arguments");

}

int cmp = 0;

if (is\_integer(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_integer > second\_val->value.mal\_integer);

}

else if (is\_integer(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_integer > second\_val->value.mal\_float);

}

else if (is\_float(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_float > second\_val->value.mal\_integer);

}

else if (is\_float(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_float > second\_val->value.mal\_float);

}

else {

/\* shouldn't happen unless new numerical type is added \*/

return make\_error\_fmt("'>': unknown numerical type");

}

if (cmp) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_greaterthanorequalto(list args) {

if (!args || !args->next || args->next->next) {

return make\_error\_fmt("'>=': expected exactly two arguments");

}

MalType\* first\_val = args->data;

MalType\* second\_val = args->next->data;

if (!is\_number(first\_val) || !is\_number(second\_val)) {

return make\_error\_fmt("'>=': expected numerical arguments");

}

int cmp = 0;

if (is\_integer(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_integer >= second\_val->value.mal\_integer);

}

else if (is\_integer(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_integer >= second\_val->value.mal\_float);

}

else if (is\_float(first\_val) && is\_integer(second\_val)) {

cmp = (first\_val->value.mal\_float >= second\_val->value.mal\_integer);

}

else if (is\_float(first\_val) && is\_float(second\_val)) {

cmp = (first\_val->value.mal\_float >= second\_val->value.mal\_float);

}

else {

/\* shouldn't happen unless new numerical type is added \*/

return make\_error\_fmt("'>=': unknown numerical type");

}

if (cmp) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_equals(list args) {

/\* Accepts any type of arguments \*/

if (!args || !args->next || args->next->next) {

return make\_error\_fmt("'=': expected exactly two arguments");

}

MalType\* first\_val = args->data;

MalType\* second\_val = args->next->data;

if (is\_sequential(first\_val) && is\_sequential(second\_val)) {

return equal\_lists(first\_val, second\_val);

}

else if (first\_val->type != second\_val->type) {

return make\_false();

}

else {

switch(first\_val->type) {

case MALTYPE\_INTEGER:

if (first\_val->value.mal\_integer == second\_val->value.mal\_integer) {

return make\_true();

} else {

return make\_false();

}

break;

case MALTYPE\_FLOAT:

if (first\_val->value.mal\_float == second\_val->value.mal\_float) {

return make\_true();

} else {

return make\_false();

}

break;

case MALTYPE\_SYMBOL:

if (strcmp(first\_val->value.mal\_symbol, second\_val->value.mal\_symbol) == 0) {

return make\_true();

} else {

return make\_false();

}

break;

case MALTYPE\_STRING:

if (strcmp(first\_val->value.mal\_string, second\_val->value.mal\_string) == 0) {

return make\_true();

} else {

return make\_false();

}

break;

case MALTYPE\_KEYWORD:

if (strcmp(first\_val->value.mal\_keyword, second\_val->value.mal\_keyword) == 0) {

return make\_true();

} else {

return make\_false();

}

break;

case MALTYPE\_HASHMAP:

return equal\_hashmaps(first\_val, second\_val);

break;

case MALTYPE\_TRUE:

case MALTYPE\_FALSE:

case MALTYPE\_NIL:

return make\_true();

break;

case MALTYPE\_FUNCTION:

if (first\_val->value.mal\_function == second\_val->value.mal\_function) {

return make\_true();

} else {

return make\_false();

}

break;

case MALTYPE\_CLOSURE:

if (&first\_val->value.mal\_closure == &second\_val->value.mal\_closure) {

return make\_true();

} else {

return make\_false();

}

break;

}

}

return make\_false();

}

MalType\* mal\_list(list args) {

/\* Accepts any number and type of arguments \*/

return make\_list(args);

}

MalType\* mal\_nth(list args) {

if (!args || !args->next || args->next->next) {

return make\_error("'nth': Expected exactly two arguments");

}

MalType\* lst = args->data;

MalType\* n = args->next->data;

if (!is\_sequential(lst)) {

return make\_error\_fmt("'nth': first argument is not a list or vector: '%s'\n", pr\_str(lst, UNREADABLY));

}

if (!is\_integer(n)) {

return make\_error\_fmt("'nth': second argument is not an integer: '%s'\n", pr\_str(lst, UNREADABLY));

}

MalType\* result = list\_nth(lst->value.mal\_list, n->value.mal\_integer);

if (result) {

return result;

}

else {

return make\_error\_fmt("'nth': index %s out of bounds for: '%s'\n", \

pr\_str(n, UNREADABLY), pr\_str(lst, UNREADABLY));

}

}

MalType\* mal\_first(list args) {

if (!args || args->next) {

return make\_error("'first': expected exactly one argument");

}

MalType\* lst = args->data;

if (!is\_sequential(lst) && !is\_nil(lst)) {

return make\_error("'first': expected a list or vector");

}

MalType\* result = list\_first(lst->value.mal\_list);

if (result) {

return result;

}

else {

return make\_nil();

}

}

MalType\* mal\_rest(list args) {

if (!args || args->next) {

return make\_error("'rest': expected exactly one argument");

}

MalType\* lst = args->data;

if (!is\_sequential(lst) && !is\_nil(lst)) {

return make\_error("'rest': expected a list or vector");

}

list result = list\_rest(lst->value.mal\_list);

if (lst) {

return make\_list(result);

}

else {

return make\_nil();

}

}

MalType\* mal\_cons(list args) {

if (!args || (args->next && args->next->next)) {

return make\_error("'cons': Expected exactly two arguments");

}

MalType\* lst = args->next->data;

if (is\_sequential(lst)) {

return make\_list(list\_push(lst->value.mal\_list, args->data));

}

else if (is\_nil(lst)) {

return make\_list(list\_push(NULL, args->data));

}

else {

return make\_error\_fmt("'cons': second argument is not a list or vector: '%s'\n", \

pr\_str(lst, UNREADABLY));

}

}

MalType\* mal\_concat(list args) {

/\* return an empty list for no arguments \*/

if (!args) {

return make\_list(NULL);

}

list new\_list = NULL;

while (args) {

MalType\* val = args->data;

/\* skip nils \*/

if (is\_nil(val)) {

args = args->next;

continue;

}

/\* concatenate lists and vectors \*/

else if (is\_sequential(val)) {

list lst = val->value.mal\_list;

new\_list = list\_concatenate(new\_list, lst);

args = args->next;

}

/\* raise an error for any non-sequence types \*/

else {

return make\_error\_fmt("'concat': all arguments must be lists or vectors '%s'", \

pr\_str(val, UNREADABLY));

}

}

return make\_list(new\_list);

}

MalType\* mal\_count(list args) {

if (args->next) {

return make\_error\_fmt("'count': too many arguments");

}

MalType\* val = args->data;

if (!is\_sequential(val) && !is\_nil(val)) {

return make\_error\_fmt("'count': argument is not a list or vector: '%s'\n", \

pr\_str(val, UNREADABLY));

}

return make\_integer(list\_count(val->value.mal\_list));

}

MalType\* mal\_list\_questionmark(list args) {

if (args->next) {

return make\_error\_fmt("'list?': too many arguments");

}

MalType\* val = args->data;

if (is\_list(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_empty\_questionmark(list args) {

if (args->next) {

return make\_error\_fmt("'empty?': too many arguments");

}

MalType\* val = args->data;

if (!is\_sequential(val)) {

return make\_error\_fmt("'empty?': argument is not a list or vector: '%s'\n", pr\_str(val, UNREADABLY));

}

if (!val->value.mal\_list) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_pr\_str(list args) {

/\* Accepts any number and type of arguments \*/

return as\_str(args, READABLY, " ");

}

MalType\* mal\_str(list args) {

/\* Accepts any number and type of arguments \*/

return as\_str(args, UNREADABLY, "");

}

MalType\* mal\_prn(list args) {

/\* Accepts any number and type of arguments \*/

return print(args, READABLY, " ");

}

MalType\* mal\_println(list args) {

/\* Accepts any number and type of arguments \*/

return print(args, UNREADABLY, " ");

}

MalType\* mal\_read\_string(list args) {

if (!args || args->next) {

return make\_error\_fmt("'read-string': expected exactly one argument");

}

MalType\* val = args->data;

if (!is\_string(val)) {

return make\_error\_fmt("'read-string': expected a string argument '%s'", pr\_str(val, UNREADABLY));

}

return read\_str(val->value.mal\_string);

}

MalType\* mal\_slurp(list args) {

if (args->next) {

return make\_error\_fmt("'slurp': too many arguments");

}

MalType\* filename = args->data;

if (!is\_string(filename)) {

return make\_error\_fmt("'slurp': expected a string argument");

}

long file\_length = 0;

FILE\* file = fopen(filename->value.mal\_string, "rb");

if (!file){

return make\_error\_fmt("'slurp': file not found '%s'", pr\_str(filename, UNREADABLY));

}

fseek(file, 0, SEEK\_END);

file\_length = ftell(file);

fseek(file, 0, SEEK\_SET);

char\* buffer = (char\*)GC\_MALLOC(sizeof(\*buffer) \* file\_length + 1);

if (file\_length != fread(buffer, sizeof(\*buffer), file\_length, file)) {

return make\_error\_fmt("'slurp': failed to read file '%s'", pr\_str(filename, UNREADABLY));

}

fclose(file);

buffer[file\_length] = '\0';

return make\_string(buffer);

}

MalType\* mal\_atom(list args) {

if (!args || args->next) {

return make\_error\_fmt("'atom': expected exactly one argument");

}

MalType\* val = args->data;

return make\_atom(val);

}

MalType\* mal\_atom\_questionmark(list args) {

if (!args || args->next) {

return make\_error\_fmt("'atom?': expected exactly one argument");

}

MalType\* val = args->data;

if (is\_atom(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_deref(list args) {

if (!args || args->next) {

return make\_error\_fmt("'deref': expected exactly one argument");

}

MalType\* val = args->data;

if (!is\_atom(val)) {

return make\_error\_fmt("'deref': value is not an atom '%s'", pr\_str(val, UNREADABLY));

}

return val->value.mal\_atom;

}

MalType\* mal\_reset\_bang(list args) {

if (!args || args->next->next) {

return make\_error\_fmt("'reset!': expected exactly two arguments");

}

MalType\* val = args->data;

if (!is\_atom(val)) {

return make\_error\_fmt("'reset!': value is not an atom '%s'", pr\_str(val, UNREADABLY));

}

val->value.mal\_atom = args->next->data;

return args->next->data;

}

MalType\* mal\_swap\_bang(list args) {

MalType\* val = args->data;

if (!is\_atom(val)) {

return make\_error\_fmt("'swap!': first argument is not an atom '%s'", pr\_str(val, UNREADABLY));

}

MalType\* fn = args->next->data;

if (!is\_callable(fn)) {

return make\_error\_fmt("'swap!': second argument is not callable '%s'", pr\_str(fn, UNREADABLY));

}

list fn\_args = args->next->next;

fn\_args = list\_push(fn\_args, val->value.mal\_atom);

MalType\* result = apply(fn, fn\_args);

if (is\_error(result)) {

return result;

}

else {

val->value.mal\_atom = result;

return result;

}

}

MalType\* mal\_throw(list args) {

if (!args || args->next) {

return make\_error\_fmt("'throw': expected exactly one argument");

}

MalType\* val = args->data;

/\* re-throw an existing exception \*/

if (is\_error(val)) {

return val;

}

/\* create a new exception \*/

else {

return wrap\_error(val);

}

}

MalType\* mal\_apply(list args) {

if (!args || !args->next) {

return make\_error("'apply': expected at least two arguments");

}

MalType\* func = args->data;

if (!is\_callable(func)) {

return make\_error("'apply': first argument must be callable");

}

/\* assemble loose arguments \*/

args = args->next;

list lst = NULL;

while(args->next) {

lst = list\_push(lst, args->data);

args = args->next;

}

MalType\* final = args->data;

if (is\_sequential(final)) {

lst = list\_concatenate(list\_reverse(lst), final->value.mal\_list);

}

else {

lst = list\_push(lst, final);

lst = list\_reverse(lst);

}

return apply(func, lst);

}

MalType\* mal\_map(list args) {

if (!args || !args->next || args->next->next) {

return make\_error("'map': expected two arguments");

}

MalType\* func = args->data;

if (!is\_callable(func)) {

return make\_error("'map': first argument must be a function");

}

MalType\* arg = args->next->data;

if (!is\_sequential(arg)) {

return make\_error("'map': second argument must be a list or vector");

}

list arg\_list = arg->value.mal\_list;

list result\_list = NULL;

while(arg\_list) {

MalType\* result = apply(func, list\_make(arg\_list->data));

/\* early return if error \*/

if (is\_error(result)) {

return result;

}

else {

result\_list = list\_push(result\_list, result);

}

arg\_list = arg\_list->next;

}

return make\_list(list\_reverse(result\_list));

}

MalType\* mal\_nil\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'nil?': expected a single argument");

}

MalType\* val = args->data;

if (is\_nil(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_true\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'true?': expected a single argument");

}

MalType\* val = args->data;

if (is\_true(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_false\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'false?': expected a single argument");

}

MalType\* val = args->data;

if (is\_false(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_symbol\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'symbol?': expected a single argument");

}

MalType\* val = args->data;

if (is\_symbol(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_symbol(list args) {

if (!args || args->next) {

return make\_error("'symbol': expected a single argument");

}

MalType\* val = args->data;

if (!is\_string(val)) {

return make\_error("'symbol': expected a string argument");

}

else {

return make\_symbol(val->value.mal\_string);

}

}

MalType\* mal\_keyword(list args) {

if (!args || args->next) {

return make\_error("'keyword': expected a single argument");

}

MalType\* val = args->data;

if (!is\_string(val) && !is\_keyword(val)) {

return make\_error("'keyword': expected a string argument");

}

else {

return make\_keyword(val->value.mal\_string);

}

}

MalType\* mal\_keyword\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'keyword?': expected a single argument");

}

MalType\* val = args->data;

if (is\_keyword(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_vec(list args) {

/\* Accepts a single argument \*/

if (!args || args->next) {

return make\_error("'vec': expected a single argument");

}

MalType\* val = args->data;

if (!is\_vector(val) && !is\_list(val) && !is\_hashmap(val)) {

return make\_error("'vec': expected a vector, list or hashmap");

}

MalType\* new\_val = copy\_type(val);

new\_val->type = MALTYPE\_VECTOR;

return new\_val;

}

MalType\* mal\_vector(list args) {

/\* Accepts any number and type of arguments \*/

return make\_vector(args);

}

MalType\* mal\_vector\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'vector?': expected a single argument");

}

MalType\* val = args->data;

if (is\_vector(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_sequential\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'sequential?': expected a single argument");

}

MalType\* val = args->data;

if (is\_sequential(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_hash\_map(list args) {

if (args && list\_count(args) % 2 == 1) {

return make\_error("'hashmap': odd number of arguments, expected key/value pairs");

}

list args\_iterator = args;

while (args\_iterator) {

MalType\* val = args\_iterator->data;

if (!is\_keyword(val) && !is\_string(val) && !is\_symbol(val)) {

return make\_error("'hashmap': keys must be keywords, symbols or strings");

}

args\_iterator = args\_iterator->next;

args\_iterator = args\_iterator->next;

}

return make\_hashmap(args);

}

MalType\* mal\_map\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'map?': expected a single argument");

}

MalType\* val = args->data;

if (is\_hashmap(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_get(list args) {

/\* TODO: implement a proper hashmap \*/

if (!args || args->next->next) {

return make\_error("'get': expected exactly two arguments");

}

MalType\* map = args->data;

if (!is\_hashmap(map) && !is\_nil(map)) {

return make\_error("'get': expected a map for the first argument");

}

MalType\* result = hashmap\_getf(map->value.mal\_list, get\_fn(args->next->data), get\_fn);

if (!result) {

return make\_nil();

}

return result;

}

MalType\* mal\_contains\_questionmark(list args) {

if (!args || args->next->next) {

return make\_error("'contains?': expected exactly two arguments");

}

MalType\* map = args->data;

if (!is\_hashmap(map)) {

return make\_error("'contains?': expected a map for the first argument");

}

MalType\* result = hashmap\_getf(map->value.mal\_list, get\_fn(args->next->data), get\_fn);

if (!result) {

return make\_false();

}

else {

return make\_true();

}

}

MalType\* mal\_assoc(list args) {

if (!args || !args->next || !args->next->next) {

return make\_error("'assoc': expected at least three arguments");

}

MalType\* map = args->data;

if (!is\_hashmap(map)) {

return make\_error("'assoc': expected a map for the first argument");

}

if (list\_count(args->next)%2 != 0) {

return make\_error("'assoc': expected even number of key/value pairs");

}

list new\_lst = list\_reverse(list\_copy(map->value.mal\_list));

args = args->next;

while (args) {

/\* try to update copy in-place \*/

hashmap result = hashmap\_updatef(new\_lst, get\_fn(args->data), args->next->data, get\_fn);

if (result) {

new\_lst = result;

}

/\* add a new key/value pair \*/

else {

new\_lst = list\_push(new\_lst,args->next->data);

new\_lst = list\_push(new\_lst,args->data);

}

args = args->next->next;

}

return make\_hashmap(new\_lst);

}

MalType\* mal\_dissoc(list args) {

if (!args || !args->next) {

return make\_error("'dissoc': expected at least two arguments");

}

MalType\* map = args->data;

if (!is\_hashmap(map)) {

return make\_error("'dissoc': expected a map for the first argument");

}

list source\_list = map->value.mal\_list;

list new\_list = NULL;

args = args->next;

while(source\_list) {

list dis\_args = args;

long dis = 0;

while(dis\_args) {

list tmp = NULL;

tmp = list\_push(tmp, source\_list->data);

tmp = list\_push(tmp, dis\_args->data);

MalType\* cmp = mal\_equals(tmp);

if (is\_true(cmp)) {

dis = 1;

break;

}

dis\_args = dis\_args->next;

}

if (!dis) {

new\_list = list\_push(new\_list, source\_list->data);

new\_list = list\_push(new\_list, source\_list->next->data);

}

source\_list = source\_list->next->next;

}

return make\_hashmap(list\_reverse(new\_list));

}

MalType\* mal\_keys(list args) {

if (!args || args->next) {

return make\_error("'keys': expected exactly one argument");

}

MalType\* map = args->data;

if (!is\_hashmap(map)) {

return make\_error("'keys': expected a map");

}

list lst = map->value.mal\_list;

if (!lst) {

return make\_list(NULL);

}

list result = list\_make(lst->data);

while(lst->next->next) {

lst = lst->next->next;

result = list\_push(result, lst->data);

}

return make\_list(result);

}

MalType\* mal\_vals(list args) {

if (!args || args->next) {

return make\_error("'vals': expected exactly one argument");

}

MalType\* map = args->data;

if (!is\_hashmap(map)) {

return make\_error("'vals': expected a map");

}

list lst = map->value.mal\_list;

if (!lst) {

return make\_list(NULL);

}

lst = lst->next;

list result = list\_make(lst->data);

while(lst->next) {

lst = lst->next->next;

result = list\_push(result, lst->data);

}

return make\_list(result);

}

MalType\* mal\_string\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'string?': expected a single argument");

}

MalType\* val = args->data;

if (is\_string(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_number\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'number?': expected a single argument");

}

MalType\* val = args->data;

if (is\_number(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_fn\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'fn?': expected a single argument");

}

MalType\* val = args->data;

if (is\_callable(val) && !is\_macro(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_macro\_questionmark(list args) {

if (!args || args->next) {

return make\_error("'macro?': expected a single argument");

}

MalType\* val = args->data;

if (is\_macro(val)) {

return make\_true();

}

else {

return make\_false();

}

}

MalType\* mal\_time\_ms(list args) {

struct timeval tv;

gettimeofday(&tv, NULL);

long ms = tv.tv\_sec \* 1000 + tv.tv\_usec/1000.0 + 0.5;

return make\_float(ms);

}

MalType\* mal\_conj(list args) {

if (!args || !args->next) {

return make\_error("'conj': Expected at least two arguments");

}

MalType\* lst = args->data;

if (!is\_sequential(lst)) {

return make\_error\_fmt("'conj': first argument is not a list or vector: '%s'\n", \

pr\_str(lst, UNREADABLY));

}

list rest = args->next;

if (is\_list(lst)) {

list new\_lst = list\_reverse(list\_copy(lst->value.mal\_list));

while(rest) {

new\_lst = list\_push(new\_lst, rest->data);

rest = rest->next;

}

return make\_list(new\_lst);

}

else /\* is\_vector(lst) \*/ {

list new\_lst = list\_copy(lst->value.mal\_list);

while(rest) {

new\_lst = list\_push(new\_lst, rest->data);

rest = rest->next;

}

return make\_vector(list\_reverse(new\_lst));

}

}

MalType\* mal\_seq(list args) {

if (!args || args->next) {

return make\_error("'seq': expected exactly one argument");

}

MalType\* val = args->data;

if (is\_sequential(val)) {

/\* empy list or vector \*/

if (!val->value.mal\_list) {

return make\_nil();

}

else {

return make\_list(val->value.mal\_list);

}

}

else if (is\_string(val)) {

/\* empty string \*/

if (\*(val->value.mal\_string) == '\0') {

return make\_nil();

}

else {

char\* ch = val->value.mal\_string;

list lst = NULL;

while(\*ch != '\0') {

char\* new\_ch = GC\_MALLOC(sizeof(\*new\_ch));

strncpy(new\_ch, ch, 1);

lst = list\_push(lst, make\_string(new\_ch));

ch++;

}

return make\_list(list\_reverse(lst));

}

}

else if (is\_nil(val)) {

return make\_nil();

}

else {

return make\_error("'seq': expected a list, vector or string");

}

}

MalType\* mal\_meta(list args) {

if (!args || args->next) {

return make\_error("'meta': expected exactly one argument");

}

MalType\* val = args->data;

if (!is\_sequential(val) && !is\_hashmap(val) && !is\_callable(val)) {

return make\_error("'meta': metadata not supported for data type");

}

if (!val->metadata) {

return make\_nil();

} else {

return val->metadata;

}

}

MalType\* mal\_with\_meta(list args) {

if (!args || !args->next || args->next->next) {

return make\_error("'with-meta': expected exactly two arguments");

}

MalType\* val = args->data;

if (!is\_sequential(val) && !is\_hashmap(val) && !is\_callable(val)) {

return make\_error("'with-meta': metadata not supported for data type");

}

MalType\* metadata = args->next->data;

MalType\* new\_val = copy\_type(val);

new\_val->metadata = metadata;

return new\_val;

}

/\* helper functions \*/

MalType\* as\_str(list args, int readably, char\* separator) {

long buffer\_length = STRING\_BUFFER\_SIZE;

long separator\_length = strlen(separator);

char\* buffer = GC\_MALLOC(sizeof(\*buffer) \* STRING\_BUFFER\_SIZE);

long char\_count = 0;

while(args) {

MalType\* arg = args->data;

char\* str = pr\_str(arg, readably);

int len = strlen(str);

char\_count += len;

char\_count += separator\_length;

if (char\_count >= buffer\_length) {

buffer = GC\_REALLOC(buffer, sizeof(\*buffer) \* char\_count + 1);

}

strncat(buffer, str, char\_count);

args = args->next;

if (args) {

strcat(buffer, separator);

}

}

return make\_string(buffer);

}

MalType\* print(list args, int readably, char\* separator) {

while(args) {

printf("%s", pr\_str(args->data, readably));

args = args->next;

if (args) {

printf("%s", separator);

}

}

printf("\n");

return make\_nil();

}

MalType\* equal\_lists(MalType\* list1, MalType\* list2) {

list first = list1->value.mal\_list;

list second = list2->value.mal\_list;

if (list\_count(first) != list\_count(second)) {

return make\_false();

}

else {

while(first && second) {

list args = NULL;

args = list\_push(args, second->data);

args = list\_push(args, first->data);

MalType\* cmp = mal\_equals(args);

if (is\_false(cmp)) {

return make\_false();

break;

}

first = first->next;

second = second->next;

}

return make\_true();

}

}

MalType\* equal\_hashmaps(MalType\* map1, MalType\* map2) {

list first = map1->value.mal\_list;

list second = map2->value.mal\_list;

if (!first && !second) {

return make\_true();

}

if (list\_count(first) != list\_count(second)) {

return make\_false();

}

while (first) {

MalType\* key1 = first->data;

MalType\* val1 = first->next->data;

MalType\* val2 = hashmap\_getf(second, get\_fn(key1), get\_fn);

if (!val2) {

return make\_false();

}

list args = NULL;

args = list\_push(args, val1);

args = list\_push(args, val2);

MalType\* cmp = mal\_equals(args);

if (is\_false(cmp)) {

return make\_false();

break;

}

first = first->next->next;

}

return make\_true();

}

/\* helper function for get \*/

char\* get\_fn(gptr data) {

MalType\* val = data;

switch (val->type) {

case MALTYPE\_STRING:

return (val->value.mal\_string);

break;

case MALTYPE\_SYMBOL:

return (val->value.mal\_symbol);

break;

case MALTYPE\_KEYWORD:

return (val->value.mal\_keyword);

break;

default:

return NULL;

}

}

#ifdef WITH\_FFI

MalType\* mal\_dot(list args) {

/\* (. "lib" "return type" "function" "arg1 type" "arg 1" ...) \*/

if (!args || !args->next || !args->next->next) {

return make\_error("'.': expected at least three arguments");

}

MalType\* lib\_name = (MalType\*)args->data;

if (!is\_string(lib\_name) && !is\_nil(lib\_name)) {

return make\_error("'.': expected library name or nil for first argument");

}

MalType\* return\_type = (MalType\*)args->next->data;

if (!is\_string(return\_type)) {

return make\_error("'.': expected string (return type) for second argument");

}

MalType\* fn\_name = (MalType\*)args->next->next->data;

if (!is\_string(fn\_name)) {

return make\_error("'.': expected string (function name) for third argument");

}

int args\_count = list\_count(args) - 3;

if (args\_count % 2 == 1) {

return make\_error("'.': expected even number of argument types and values");

}

list arg\_types\_list = NULL;

list arg\_vals\_list = NULL;

args = args->next->next->next;

while(args) {

MalType\* val\_type = (MalType\*)args->data;

MalType\* val = (MalType\*)args->next->data;

if (!is\_string(val\_type)) {

return make\_error\_fmt("'.': expected strings for argument types: '%s'", pr\_str(val\_type, UNREADABLY));

}

arg\_types\_list = list\_push(arg\_types\_list, val\_type);

arg\_vals\_list = list\_push(arg\_vals\_list, val);

args = args->next->next;

}

arg\_types\_list = list\_reverse(arg\_types\_list);

arg\_vals\_list = list\_reverse(arg\_vals\_list);

/\* open a shared library dynamically and get hold of a function \*/

gptr lib\_handle;

if (!is\_nil(lib\_name)) {

lib\_handle = dlopen(lib\_name->value.mal\_string, RTLD\_LAZY);

} else {

lib\_handle = dlopen(NULL, RTLD\_LAZY);

}

if (!lib\_handle) {

return make\_error\_fmt("'ffi`' reports: %s", dlerror());

}

gptr fn = dlsym(lib\_handle, fn\_name->value.mal\_string);

char\* error;

if ((error = dlerror()) != NULL) {

return make\_error\_fmt("'ffi' dlsym could not get handle to function '%s': %s", fn\_name->value.mal\_string, error);

}

/\* use libffi to call function \*/

ffi\_cif cif;

ffi\_type\* ret\_type;

ffi\_type\* arg\_types[20];

void\* arg\_vals[20];

ffi\_status status;

ffi\_type\* ffi\_get\_type(char \*type, MalType\* err);

MalType\* mal\_err = make\_nil();

/\* set return type \*/

MalType\* make\_type(char \*type);

MalType\* retval = make\_type(return\_type->value.mal\_string);

ret\_type = ffi\_get\_type(return\_type->value.mal\_string, mal\_err);

if (is\_error(mal\_err)) { return mal\_err; }

int arg\_count = list\_count(arg\_types\_list);

/\* Set the argument types and values \*/

for (int i = 0; i < arg\_count; i++) {

MalType\* val\_type = (MalType\*)arg\_types\_list->data;

arg\_types[i] = ffi\_get\_type(val\_type->value.mal\_string, mal\_err);

if (is\_error(mal\_err)) { return mal\_err; }

MalType\* val = (MalType\*)arg\_vals\_list->data;

arg\_vals[i] = &(val->value);

arg\_types\_list = arg\_types\_list->next;

arg\_vals\_list = arg\_vals\_list->next;

}

/\* perform the call \*/

status = ffi\_prep\_cif(&cif, FFI\_DEFAULT\_ABI, arg\_count, ret\_type, arg\_types);

if (status != FFI\_OK) {

return make\_error\_fmt("'ffi' call to ffi\_prep\_cif failed with code: %d\n", status);

}

ffi\_call(&cif, FFI\_FN(fn), &retval->value, arg\_vals);

/\* close the library \*/

dlclose(lib\_handle);

if (ret\_type == &ffi\_type\_void) {

return make\_nil();

} else {

return retval;

}

}

/\* helper function for ffi \*/

ffi\_type\* ffi\_get\_type(char \*type, MalType\* err) {

if ((strcmp("void", type) == 0)) {

return &ffi\_type\_void;

}

else if ((strcmp("string", type) == 0) ||

(strcmp("char\*", type) == 0) ||

(strcmp("char \*", type) == 0)) {

return &ffi\_type\_pointer;

}

else if ((strcmp("integer", type) == 0) ||

(strcmp("int64", type) == 0)) {

return &ffi\_type\_sint64;

}

else if ((strcmp("int32", type) == 0)) {

return &ffi\_type\_sint32;

}

else if (strcmp("double", type) == 0) {

return &ffi\_type\_double;

}

else if (strcmp("float", type) == 0) {

return &ffi\_type\_float;

}

else {

err = make\_error\_fmt("'ffi' type not recognised '%'", type);

return NULL;

}

}

/\* helper function for ffi \*/

MalType\* make\_type(char \*type) {

if ((strcmp("void", type) == 0)) {

return make\_nil();

}

else if ((strcmp("string", type) == 0) ||

(strcmp("char\*", type) == 0) ||

(strcmp("char \*", type) == 0)) {

return make\_string("");

}

else if ((strcmp("integer", type) == 0) ||

(strcmp("int64", type) == 0)) {

return make\_integer(0);

}

else if ((strcmp("int32", type) == 0)) {

return make\_integer(0);

}

else if (strcmp("double", type) == 0) {

return make\_float(0);

}

else if (strcmp("float", type) == 0) {

return make\_float(0);

}

else {

return make\_error\_fmt("'ffi' type not supported '%s'", type);

}

}

#endif

***reader.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <gc.h>

#include "reader.h"

#define TOKEN\_SPECIAL\_CHARACTER 1

#define TOKEN\_STRING 2

#define TOKEN\_INTEGER 3

#define TOKEN\_FLOAT 4

#define TOKEN\_SYMBOL 5

#define TOKEN\_COMMENT 6

#define TOKEN\_KEYWORD 7

#define TOKEN\_TRUE 8

#define TOKEN\_FALSE 9

#define TOKEN\_NIL 10

#define SYMBOL\_NIL "nil"

#define SYMBOL\_TRUE "true"

#define SYMBOL\_FALSE "false"

#define SYMBOL\_QUOTE "quote"

#define SYMBOL\_QUASIQUOTE "quasiquote"

#define SYMBOL\_UNQUOTE "unquote"

#define SYMBOL\_SPLICE\_UNQUOTE "splice-unquote"

#define SYMBOL\_DEREF "deref"

#define SYMBOL\_WITH\_META "with-meta"

Reader\* reader\_make(long token\_capacity) {

Reader\* reader = GC\_MALLOC(sizeof(\*reader));

reader->max\_tokens = token\_capacity;

reader->position = 0;

reader->token\_count = 0;

reader->token\_data = GC\_MALLOC(sizeof(Token\*) \* token\_capacity);

reader->error = NULL;

return reader;

}

Reader\* reader\_append(Reader\* reader, Token\* token) {

if (reader->token\_count < reader->max\_tokens) {

reader->token\_data[reader->token\_count] = token;

reader->token\_count++;

}

else {

/\* TODO: expand the storage more intelligently \*/

reader->max\_tokens \*= 2;

reader = GC\_REALLOC(reader, sizeof(\*reader) \* reader->max\_tokens);

reader->token\_data[reader->token\_count] = token;

reader->token\_count++;

}

return reader;

}

Token\* reader\_peek(const Reader\* reader) {

return (reader->token\_data[reader->position]);

}

Token\* reader\_next(Reader\* reader) {

Token\* tok = reader->token\_data[reader->position];

if (reader->position == -1) {

return NULL;

}

else if (reader->position < reader->token\_count) {

(reader->position)++;

return tok;

}

else {

reader->position = -1;

return tok;

}

}

void reader\_print(Reader\* reader) {

/\* NOTE: needed for debugging the reader only \*/

Token\* tok;

for (long i = 0; i < reader->token\_count; i++) {

tok = reader\_next(reader);

switch (tok->type) {

case TOKEN\_SPECIAL\_CHARACTER:

printf("special character: %s", tok->data);

break;

case TOKEN\_STRING:

printf("string: %s", tok->data);

break;

case TOKEN\_INTEGER:

printf("integer: %s", tok->data);

break;

case TOKEN\_FLOAT:

printf("float: %s", tok->data);

break;

case TOKEN\_SYMBOL:

printf("symbol: %s", tok->data);

break;

case TOKEN\_COMMENT:

printf("comment: \"%s\"", tok->data);

break;

case TOKEN\_KEYWORD:

printf("keyword: %s", tok->data);

break;

case TOKEN\_TRUE:

printf("true: %s", tok->data);

break;

case TOKEN\_FALSE:

printf("false: %s", tok->data);

break;

case TOKEN\_NIL:

printf("nil: %s", tok->data);

break;

}

/\* print an error for any tokens with an error string \*/

tok->error ? printf(" - %s", tok->error) : 0;

}

}

MalType\* read\_str(char\* token\_string) {

Reader\* reader = tokenize(token\_string);

if (reader->error) {

return make\_error\_fmt("Reader error: %s", reader->error);

}

else if (reader->token\_count == 0) {

return make\_nil();

}

else {

return read\_form(reader);

}

}

Reader\* tokenize(char\* token\_string) {

/\* allocate enough space for a Reader \*/

/\* TODO: over-allocates space \*/

Reader\* reader = reader\_make(strlen(token\_string));

for (char\* next = token\_string; \*next != '\0';) {

Token\* token = NULL;

switch (\*next) {

/\* skip whitespace \*/

case ' ':

case ',':

case 0x0A: /\* newline \*/

next++;

token = NULL; /\* no token for whitespace \*/

break;

/\* single character token \*/

case '[':

case '\\':

case ']':

case '{':

case '}':

case '(':

case ')':

case '\'':

case '@':

case '`':

case '^':

next = read\_fixed\_length\_token(next, &token, 1);

break;

/\* single or double character token \*/

case '~':

if ( \*(next + 1) == '@' ) {

next = read\_fixed\_length\_token(next, &token, 2);

}

else {

next = read\_fixed\_length\_token(next, &token, 1);

}

break;

/\* read string of characters within double quotes \*/

case '"':

next = read\_string\_token(next, &token);

break;

/\* read a comment - all remaining input until newline \*/

case ';':

next = read\_comment\_token(next, &token);

token = NULL; /\* skip token for comments \*/

break;

/\* read an integer \*/

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

next = read\_number\_token(next, &token);

// next = read\_integer\_token(next, &token);

break;

/\* integer may be prefixed with +/- \*/

case '+':

case '-':

if (isdigit(next[1])) {

next = read\_number\_token(next, &token);

// next = read\_integer\_token(next, &token);

}

else { /\* if not digits it is part of a symbol \*/

next = read\_symbol\_token(next, &token);

}

break;

/\* read keyword \*/

case ':':

next = read\_keyword\_token(next, &token);

break;

/\* read anything else as a symbol \*/

default:

next = read\_symbol\_token(next, &token);

break;

}

if (!token) {

/\* if no token was read (whitespace or comments)

continue the loop \*/

continue;

}

else {

if (token->error) {

/\* report any errors with an early return \*/

reader = reader\_append(reader, token);

reader->error = token->error;

return reader;

}

/\* otherwise append the token and continue \*/

reader = reader\_append(reader, token);

}

}

return reader;

}

char\* read\_fixed\_length\_token(char\* current, Token\*\* ptoken, int n) {

\*ptoken = token\_allocate(current, n, TOKEN\_SPECIAL\_CHARACTER, NULL);

return (current + n);

}

char\* read\_terminated\_token (char\* current, Token\*\* ptoken, int token\_type) {

static char\* const terminating\_characters = " ,[](){};\n";

/\* search for first terminating character \*/

char\* end = strpbrk(current, terminating\_characters);

/\* if terminating character is not found it implies the end of the string \*/

long token\_length = !end ? strlen(current) : (end - current);

/\* next token starts with the terminating character \*/

\*ptoken = token\_allocate(current, token\_length, token\_type, NULL);

return (current + token\_length);

}

char\* read\_symbol\_token (char\* current, Token\*\* ptoken) {

char\* next = read\_terminated\_token(current, ptoken, TOKEN\_SYMBOL);

/\* check for reserved symbols \*/

if (strcmp((\*ptoken)->data, SYMBOL\_NIL) == 0) {

(\*ptoken)->type = TOKEN\_NIL;

}

else if (strcmp((\*ptoken)->data, SYMBOL\_TRUE) == 0) {

(\*ptoken)->type = TOKEN\_TRUE;

}

else if (strcmp((\*ptoken)->data, SYMBOL\_FALSE) == 0) {

(\*ptoken)->type = TOKEN\_FALSE;

}

/\* TODO: check for invalid characters \*/

return next;

}

char\* read\_keyword\_token (char\* current, Token\*\* ptoken) {

/\* TODO: check for invalid characters \*/

return read\_terminated\_token(current + 1, ptoken, TOKEN\_KEYWORD);

}

char\* read\_number\_token(char\* current, Token\*\* ptoken) {

int has\_decimal\_point = 0;

char\* next = read\_terminated\_token(current, ptoken, TOKEN\_INTEGER);

long token\_length = next - current;

/\* first char is either digit or '+' or '-'

check the rest consists of valid characters \*/

for (long i = 1; i < token\_length; i++) {

if ((\*ptoken)->data[i] == '.' && has\_decimal\_point) {

(\*ptoken)->error = "Invalid character reading number";

break;

}

else if ((\*ptoken)->data[i] == '.' && !has\_decimal\_point) {

has\_decimal\_point = 1;

(\*ptoken)->type = TOKEN\_FLOAT;

break;

}

else if (!(isdigit((\*ptoken)->data[i]))) {

(\*ptoken)->error = "Invalid character reading number";

break;

}

}

return next;

}

char\* read\_string\_token(char\* current, Token\*\* ptoken) {

char \*start, \*end, \*error = NULL;

long token\_length = 0;

start = current + 1;

while(1) {

end = strchr(start, '"'); /\* find the next " character \*/

/\* handle failure to find closing quotes - implies end of input has been reached \*/

if (!end) {

end = current + strlen(current);

token\_length = strlen(current);

error = "EOF reached with unterminated string";

break;

}

/\* if the character preceding the " is a '\' character (escape), need to check if it is escaping the " and if it

is then keep searching from the next character \*/

else if ( \*(end - 1) == '\\') {

char\* back\_ptr = end - 1;

while (\*back\_ptr == '\\') {

back\_ptr--; /\* back up to count the escape characters '\' \*/

}

long escape\_chars = (end - 1) - back\_ptr;

if (escape\_chars % 2 == 1) { /\* odd number of '\' chars means " is not quoted \*/

start = end + 1; /\* so keep searching \*/

} else {

/\* even number of '\' characters means we found the terminating quote mark \*/

token\_length = (end - current - 1); /\* quotes are excluded from string token \*/

break;

}

}

else {

token\_length = (end - current - 1); /\* quotes are excluded from string token \*/

break;

}

}

char\* unescaped\_string = unescape\_string(current + 1, token\_length);

\*ptoken = token\_allocate(unescaped\_string, strlen(unescaped\_string), TOKEN\_STRING, error);

return (end + 1);

}

char\* read\_comment\_token(char\* current, Token\*\* ptoken) {

/\* comment includes all remaining characters to the next newline \*/

/\* search for newline character \*/

char\* end = strchr(current, 0x0A);

/\* if newline is not found it implies the end of string is reached \*/

long token\_chars = !end ? strlen(current) : (end - current);

\*ptoken = token\_allocate(current, token\_chars, TOKEN\_COMMENT, NULL);

return (current + token\_chars + 1); /\* next token starts with the char after the newline \*/

}

MalType\* read\_form(Reader\* reader) {

if (reader->token\_count > 0) {

Token\* tok = reader\_peek(reader);

if (tok->type == TOKEN\_SPECIAL\_CHARACTER) {

switch(tok->data[0]) {

case '(':

return read\_list(reader);

break;

case '[':

return read\_vector(reader);

break;

case '{':

return read\_hashmap(reader);

break;

case '\'':

/\* create and return a MalType list (quote read\_form) \*/

return make\_symbol\_list(reader, SYMBOL\_QUOTE);

break;

case '`':

/\* create and return a MalType list (quasiquote read\_form) \*/

return make\_symbol\_list(reader, SYMBOL\_QUASIQUOTE);

break;

case '~':

if (tok->data[1] == '@') {

/\* create and return a MalType list (splice-unquote read\_form) \*/

return make\_symbol\_list(reader, SYMBOL\_SPLICE\_UNQUOTE);

}

else {

/\* create and return a MalType list (unquote read\_form) \*/

return make\_symbol\_list(reader, SYMBOL\_UNQUOTE);

}

case '@':

/\* create and return a MalType list (deref read\_form) \*/

return make\_symbol\_list(reader, SYMBOL\_DEREF);

case '^':

/\* create and return a MalType list (with-meta <second-form> <first-form>

where first form should ne a metadata map and second form is somethingh

that can have metadata attached \*/

reader\_next(reader);

/\* grab the components of the list \*/

MalType\* symbol = make\_symbol(SYMBOL\_WITH\_META);

MalType\* first\_form = read\_form(reader);

MalType\* second\_form = read\_form(reader);

/\* push the symbol and the following forms onto a list \*/

list lst = NULL;

lst = list\_push(lst, symbol);

lst = list\_push(lst, second\_form);

lst = list\_push(lst, first\_form);

lst = list\_reverse(lst);

return make\_list(lst);

default:

/\* shouldn't happen \*/

return make\_error\_fmt("Reader error: Unknown special character '%c'", tok->data[0]);

}

} else { /\* Not a special character \*/

return read\_atom(reader);

}

}

else { /\* no tokens \*/

return NULL;

}

}

MalType\* read\_list(Reader\* reader) {

MalType\* retval = read\_matched\_delimiters(reader, '(', ')' );

if (is\_error(retval)) {

retval = make\_error("Reader error: unbalanced parenthesis '()'");

}

else {

retval->type = MALTYPE\_LIST;

}

return retval;

}

MalType\* read\_vector(Reader\* reader) {

MalType\* retval = read\_matched\_delimiters(reader, '[', ']' );

if (is\_error(retval)) {

retval = make\_error("Reader error: unbalanced brackets '[]'");

}

else {

retval->type = MALTYPE\_VECTOR;

}

return retval;

}

MalType\* read\_hashmap(Reader\* reader) {

MalType\* retval = read\_matched\_delimiters(reader, '{', '}' );

if (is\_error(retval)) {

retval = make\_error("Reader error: unbalanced braces '{}'");

}

else if (list\_count(retval->value.mal\_list)%2 != 0) {

retval = make\_error("Reader error: missing value in map literal");

}

else {

retval->type = MALTYPE\_HASHMAP;

}

return retval;

}

MalType\* read\_matched\_delimiters(Reader\* reader, char start\_delimiter, char end\_delimiter) {

/\* TODO: separate implementation of hashmap and vector \*/

Token\* tok = reader\_next(reader);

list lst = NULL;

if (reader\_peek(reader)->data[0] == end\_delimiter) {

reader\_next(reader);

return make\_list(NULL);

}

else {

while (tok->data[0] != end\_delimiter) {

MalType\* val = read\_form(reader);

lst = list\_push(lst, (gptr)val);

tok = reader\_peek(reader);

if (!tok) {

/\* unbalanced parentheses \*/

return make\_error("");

}

}

reader\_next(reader);

return make\_list(list\_reverse(lst));

}

}

MalType\* read\_atom(Reader\* reader) {

Token\* tok = reader\_next(reader);

switch (tok->type) {

case TOKEN\_SPECIAL\_CHARACTER:

return make\_symbol(tok->data);

break;

case TOKEN\_COMMENT:

return make\_error("Error: comment found in token strea");

break;

case TOKEN\_STRING:

return make\_string(tok->data);

break;

case TOKEN\_INTEGER:

return make\_integer(strtol(tok->data, NULL, 10));

break;

case TOKEN\_FLOAT:

return make\_float(atof(tok->data));

break;

case TOKEN\_SYMBOL:

return make\_symbol(tok->data);

break;

case TOKEN\_KEYWORD:

return make\_keyword(tok->data);

break;

case TOKEN\_TRUE:

return make\_true();

break;

case TOKEN\_FALSE:

return make\_false();

break;

case TOKEN\_NIL:

return make\_nil();

break;

}

return make\_error("Reader error: Unknown atom type");

}

MalType\* make\_symbol\_list(Reader\* reader, char\* symbol\_name) {

reader\_next(reader);

list lst = NULL;

/\* push the symbol and the following form onto the list \*/

lst = list\_push(lst, make\_symbol(symbol\_name));

lst = list\_push(lst, read\_form(reader));

return make\_list(list\_reverse(lst));

}

Token\* token\_allocate(char\* str, long num\_chars, int type, char\* error) {

/\* allocate space for the string \*/

char\* data = GC\_MALLOC(sizeof(\*data) \* num\_chars + 1); /\* include space for null byte \*/

strncpy (data, str, num\_chars); /\* copy num\_chars characters into data \*/

data[num\_chars] = '\0'; /\* manually add the null byte \*/

/\* allocate space for the token struct \*/

Token\* token = GC\_MALLOC(sizeof(\*token));

token->data = data;

token->type = type;

token->error = error;

return token;

}

char\* unescape\_string(char\* str, long length) {

char\* dest = GC\_MALLOC(sizeof(\*dest)\*length + 1);

long j = 0;

for (long i = 0; i < length; i++) {

/\* look for the quoting character \*/

if (str[i] == '\\') {

switch (str[i+1]) {

/\* replace '\"' with normal '"' \*/

case '"':

dest[j++]='"';

i++; /\* skip extra char \*/

break;

/\* replace '\n' with newline 0x0A \*/

case 'n':

dest[j++]= 0x0A;

i++; /\* skip extra char \*/

break;

/\* replace '\\' with '\' \*/

case '\\':

dest[j++]= '\\';

i++; /\* skip extra char \*/

break;

default:

/\* just a '\' symbol so copy it \*/

dest[j++]='\\';

}

}

/\* not a quote so copy it \*/

else {

dest[j++] = str[i];

}

}

dest[j] = '\0';

return dest;

}

***env.c***

#include <stdio.h>

#include <gc.h>

#include "libs/hashmap/hashmap.h"

#include "types.h"

#include "env.h"

#include "reader.h"

/\* Note: caller must make sure enough exprs to match symbols \*/

Env\* env\_make(Env\* outer, list symbol\_list, list exprs\_list, MalType\* more\_symbol) {

Env\* env = GC\_MALLOC(sizeof(\*env));

env->outer = outer;

env->data = NULL;

while (symbol\_list) {

env = env\_set(env, symbol\_list->data, exprs\_list->data);

symbol\_list = symbol\_list->next;

exprs\_list = exprs\_list->next;

}

/\* set the 'more' symbol if there is one \*/

if (more\_symbol) {

env = env\_set(env, more\_symbol, make\_list(exprs\_list));

}

return env;

}

Env\* env\_set(Env\* current, MalType\* symbol, MalType\* value) {

current->data = hashmap\_put(current->data, symbol->value.mal\_symbol, value);

return current;

}

Env\* env\_find(Env\* current, MalType\* symbol) {

MalType\* val = hashmap\_get(current->data, symbol->value.mal\_symbol);

if (val) {

return current;

}

else if (current->outer) {

return env\_find(current->outer, symbol);

}

else {

return NULL; /\* not found \*/

}

}

MalType\* env\_get(Env\* current, MalType\* symbol) {

Env\* env = env\_find(current, symbol);

if (env) {

return hashmap\_get(env->data, symbol->value.mal\_symbol);

}

else {

return make\_error\_fmt("'%s' not found", symbol->value.mal\_symbol);

}

}

Env\* env\_set\_C\_fn(Env\* current, char\* symbol\_name, MalType\*(\*fn)(list)) {

return env\_set(current, make\_symbol(symbol\_name), make\_function(fn));

}

***printer.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include "printer.h"

#define PRINT\_NIL "nil"

#define PRINT\_TRUE "true"

#define PRINT\_FALSE "false"

#define INTEGER\_BUFFER\_SIZE 16

#define SYMBOL\_BUFFER\_SIZE 32

#define FUNCTION\_BUFFER\_SIZE 256

#define STRING\_BUFFER\_SIZE 256

#define LIST\_BUFFER\_SIZE 1024

char\* pr\_str(MalType\* val, int readably) {

if (!val) {

return "";

}

switch(val->type) {

case MALTYPE\_SYMBOL:

return snprintfbuf(SYMBOL\_BUFFER\_SIZE, "%s", val->value.mal\_symbol);

break;

case MALTYPE\_KEYWORD:

return snprintfbuf(SYMBOL\_BUFFER\_SIZE, ":%s", val->value.mal\_keyword);

break;

case MALTYPE\_INTEGER:

return snprintfbuf(SYMBOL\_BUFFER\_SIZE, "%ld", val->value.mal\_integer);

break;

case MALTYPE\_FLOAT:

return snprintfbuf(SYMBOL\_BUFFER\_SIZE, "%lf", val->value.mal\_float);

break;

case MALTYPE\_STRING:

if (readably) {

return snprintfbuf(STRING\_BUFFER\_SIZE, "%s", escape\_string(val->value.mal\_string));

}

else {

return snprintfbuf(STRING\_BUFFER\_SIZE, "%s",val->value.mal\_string);

}

break;

case MALTYPE\_TRUE:

return PRINT\_TRUE;

break;

case MALTYPE\_FALSE:

return PRINT\_FALSE;

break;

case MALTYPE\_NIL:

return PRINT\_NIL;

break;

case MALTYPE\_LIST:

return pr\_str\_list(val->value.mal\_list, readably, "(", ")", " ");

break;

case MALTYPE\_VECTOR:

return pr\_str\_list(val->value.mal\_list, readably, "[", "]", " ");

break;

case MALTYPE\_HASHMAP:

return pr\_str\_list(val->value.mal\_list, readably, "{", "}", " ");

break;

case MALTYPE\_FUNCTION:

return snprintfbuf(FUNCTION\_BUFFER\_SIZE, "#<function::native@%p>", val->value.mal\_function);

break;

case MALTYPE\_CLOSURE:

{

MalType\* definition = (val->value.mal\_closure)->definition;

MalType\* parameters = (val->value.mal\_closure)->parameters;

MalType\* more\_symbol = (val->value.mal\_closure)->more\_symbol;

list lst = parameters->value.mal\_list;

if (more\_symbol) {

lst = list\_reverse(lst);

lst = list\_push(lst, make\_symbol(snprintfbuf(STRING\_BUFFER\_SIZE, "%s%s", "&", more\_symbol->value.mal\_symbol)));

lst = list\_reverse(lst);

}

if (val->is\_macro) {

return snprintfbuf(FUNCTION\_BUFFER\_SIZE, "#<function::macro: (fn\* %s %s))", \

pr\_str(make\_list(lst), UNREADABLY), pr\_str(definition, UNREADABLY));

}

else {

return snprintfbuf(FUNCTION\_BUFFER\_SIZE, "#<function::closure: (fn\* %s %s))", \

pr\_str(make\_list(lst), UNREADABLY), pr\_str(definition, UNREADABLY));

}

}

break;

case MALTYPE\_ATOM:

return snprintfbuf(STRING\_BUFFER\_SIZE, "(atom %s)", pr\_str(val->value.mal\_atom, readably));

break;

case MALTYPE\_ERROR:

return snprintfbuf(STRING\_BUFFER\_SIZE, "Uncaught error: %s", pr\_str(val->value.mal\_error, UNREADABLY));

break;

default:

/\* can't happen unless a new MalType is added \*/

return "Printer error: unknown type\n";

break;

}

}

char\* pr\_str\_list(list lst, int readably, char\* start\_delimiter, char\* end\_delimiter, char\* separator) {

char\* list\_buffer = GC\_MALLOC(sizeof(\*list\_buffer) \* LIST\_BUFFER\_SIZE);

long buffer\_length = LIST\_BUFFER\_SIZE;

/\* add the start delimiter \*/

list\_buffer = strcpy(list\_buffer, start\_delimiter);

long len = strlen(start\_delimiter);

long count = len;

while (lst) {

/\* concatenate next element \*/

MalType\* data = lst->data;

char\* str = pr\_str(data, readably);

len = strlen(str);

count += len;

if (count >= buffer\_length) {

buffer\_length += (count + 1);

list\_buffer = GC\_REALLOC(list\_buffer, buffer\_length);

}

strncat(list\_buffer, str, len);

lst = lst->next;

if (lst) {

len = strlen(separator);

count += len;

if (count >= buffer\_length) {

buffer\_length += (count + 1);

list\_buffer = GC\_REALLOC(list\_buffer, buffer\_length);

}

/\* add the separator \*/

strncat(list\_buffer, separator, len);

}

}

if (count >= buffer\_length) {

len = strlen(end\_delimiter);

count += len;

buffer\_length += (count + 1);

list\_buffer = GC\_REALLOC(list\_buffer, buffer\_length);

}

/\* add the end delimiter \*/

strncat(list\_buffer, end\_delimiter, len);

return list\_buffer;

}

char\* escape\_string(char\* str) {

long buffer\_length = 2\*(strlen(str) + 1) ; /\* allocate a reasonable initial buffer size \*/

char\* buffer = GC\_MALLOC(sizeof(\*buffer) \* buffer\_length);

strcpy(buffer,"\"");

char\* curr = str;

while(\*curr != '\0') {

switch (\*curr) {

case '"':

strcat(buffer, "\\\"");

break;

case '\\':

strcat(buffer, "\\\\");

break;

case 0x0A:

strcat(buffer, "\\n");

break;

default:

strncat(buffer, curr, 1);

}

curr++;

/\* check for overflow and increase buffer size \*/

if ((curr - str) >= buffer\_length) {

buffer\_length \*= 2;

buffer = GC\_REALLOC(buffer, sizeof(\*buffer) \* buffer\_length);

}

}

strcat(buffer, "\"");

/\* trim the buffer to the size of the actual escaped string \*/

buffer\_length = strlen(buffer);

buffer = GC\_REALLOC(buffer, sizeof(\*buffer) \* buffer\_length + 1);

return buffer;

}

char\* snprintfbuf(long initial\_size, char\* fmt, ...) {

/\* this is just a wrapper for the \*printf family that ensures the

string is long enough to hold the contents \*/

va\_list argptr;

va\_start(argptr, fmt);

char\* buffer = GC\_MALLOC(sizeof(\*buffer) \* initial\_size);

long n = vsnprintf(buffer, initial\_size, fmt, argptr);

va\_end(argptr);

if (n > initial\_size) {

va\_start(argptr, fmt);

buffer = GC\_REALLOC(buffer, sizeof(\*buffer) \* n);

vsnprintf(buffer, n, fmt, argptr);

va\_end(argptr);

}

return buffer;

}

***step5\_tco.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#include "core.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define SYMBOL\_IF "if"

#define SYMBOL\_FNSTAR "fn\*"

#define SYMBOL\_DO "do"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\*\* env);

void eval\_letstar(MalType\*\* ast, Env\*\* env);

void eval\_if(MalType\*\* ast, Env\*\* env);

MalType\* eval\_fnstar(MalType\* ast, Env\* env);

MalType\* eval\_do(MalType\* ast, Env\* env);

/\* Use goto to jump here rather than calling eval for tail-call elimination \*/

TCE\_entry\_point:

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_letstar(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_IF) == 0) {

/\* TCE - modify ast directly and jump back to eval \*/

eval\_if(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_FNSTAR) == 0) {

return eval\_fnstar(ast, env);

}

else if (strcmp(symbol, SYMBOL\_DO) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

ast = eval\_do(ast, env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else if (is\_closure(func)) {

MalClosure\* closure = func->value.mal\_closure;

list params = (closure->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(evlst->next);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !closure->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

/\* TCE - modify ast and env directly and jump back to eval \*/

env = env\_make(closure->env, params, evlst->next, closure->more\_symbol);

ast = func->value.mal\_closure->definition;

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

else {

return make\_error\_fmt("first item in list is not callable: '%s'", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

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

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.5\n");

puts("Press Ctrl+d to exit\n");

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

ns\* core = ns\_make\_core();

hashmap mappings = core->mappings;

while (mappings) {

char\* symbol = mappings->data;

MalType\*(\*function)(list) = (MalType\*(\*)(list))mappings->next->data;

env\_set\_C\_fn(repl\_env, symbol, function);

/\* pop symbol and function from hashmap/list \*/

mappings = mappings->next->next;

}

/\* add not function \*/

/\* not using rep as it prints the result \*/

EVAL(READ("(def! not (fn\* (a) (if a false true)))"), repl\_env);

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'def!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'def!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)){

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

void eval\_letstar(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_error("'let\*': missing bindings list");

return;

}

MalType\* bindings = lst->next->data;

MalType\* forms = lst->next->next ? lst->next->next->data : make\_nil();

if (!is\_sequential(bindings)) {

\*ast = make\_error("'let\*': first argument is not list or vector");

return;

}

list bindings\_list = bindings->value.mal\_list;

if (list\_count(bindings\_list) % 2 == 1) {

\*ast = make\_error("'let\*': expected an even number of binding pairs");

return;

}

Env\* letstar\_env = env\_make(\*env, NULL, NULL, NULL);

/\* evaluate the bindings \*/

while(bindings\_list) {

MalType\* symbol = bindings\_list->data;

MalType\* value = EVAL(bindings\_list->next->data, letstar\_env);

/\* early return from error \*/

if (is\_error(value)) {

\*ast = value;

return;

}

env\_set(letstar\_env, symbol, value);

bindings\_list = bindings\_list->next->next;

}

\*env = letstar\_env;

\*ast = forms;

return;

}

void eval\_if(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next || !lst->next->next) {

\*ast = make\_error("'if': too few arguments");

return;

}

if (lst->next->next->next && lst->next->next->next->next) {

\*ast = make\_error("'if': too many arguments");

return;

}

MalType\* condition = EVAL(lst->next->data, \*env);

if (is\_error(condition)) {

\*ast = condition;

return;

}

if (is\_false(condition) || is\_nil(condition)) {

/\* check whether false branch is present \*/

if (lst->next->next->next) {

\*ast = lst->next->next->next->data;

return;

}

else {

\*ast = make\_nil();

return;

}

} else {

\*ast = lst->next->next->data;

return;

}

}

MalType\* eval\_fnstar(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* regularise\_parameters(list\* params, MalType\*\* more);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'fn\*': missing argument list");

}

else if (!lst->next->next) {

return make\_error("'fn\*': missing function body");

}

MalType\* params = lst->next->data;

list params\_list = params->value.mal\_list;

MalType\* more\_symbol = NULL;

MalType\* result = regularise\_parameters(&params\_list, &more\_symbol);

if (is\_error(result)) { return result; }

MalType\* definition = lst->next->next->data;

MalType\* regular\_params = make\_list(params\_list);

return make\_closure(env, regular\_params, definition, more\_symbol);

}

MalType\* eval\_do(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

/\* handle empty 'do' \*/

if (!lst->next) {

return make\_nil();

}

/\* evaluate all but the last form \*/

lst = lst->next;

while (lst->next) {

MalType\* val = EVAL(lst->data, env);

/\* return error early \*/

if (is\_error(val)) {

return val;

}

lst = lst->next;

}

/\* return the last form for TCE evaluation \*/

return lst->data;

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* regularise\_parameters(list\* args, MalType\*\* more\_symbol) {

/\* forward reference \*/

char\* symbol\_fn(gptr data);

list regular\_args = NULL;

while (\*args) {

MalType\* val = (\*args)->data;

if (!is\_symbol(val)) {

return make\_error\_fmt("non-symbol found in fn argument list '%s'", \

pr\_str(val, UNREADABLY));

}

if (val->value.mal\_symbol[0] == '&') {

/\* & is found but there is no symbol \*/

if (val->value.mal\_symbol[1] == '\0' && !(\*args)->next) {

return make\_error("missing symbol after '&' in argument list");

}

/\* & is found and there is a single symbol after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next &&

is\_symbol((\*args)->next->data) && !(\*args)->next->next)) {

/\* TODO: check symbol is no a duplicate of one already on the list \*/

\*more\_symbol = (\*args)->next->data;

break;

}

/\* & is found and there extra symbols after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next && (\*args)->next->next)) {

return make\_error\_fmt("unexpected symbol after '& %s' in argument list: '%s'", \

pr\_str((\*args)->next->data, UNREADABLY), \

pr\_str((\*args)->next->next->data, UNREADABLY));

}

/\* & is found as part of the symbol and no other symbols \*/

else if (val->value.mal\_symbol[1] != '\0' && !(\*args)->next) {

\*more\_symbol = make\_symbol((val->value.mal\_symbol + 1));

break;

}

/\* & is found as part of the symbol but there are other symbols after \*/

else if (val->value.mal\_symbol[1] != '\0' && (\*args)->next) {

return make\_error\_fmt("unexpected symbol after '%s' in argument list: '%s'", \

pr\_str(val, UNREADABLY), \

pr\_str((\*args)->next->data, UNREADABLY));

}

}

/\* & is not found - add the symbol to the regular argument list \*/

else {

if (list\_findf(regular\_args, val->value.mal\_symbol, symbol\_fn) > 0) {

return make\_error\_fmt("duplicate symbol in argument list: '%s'", pr\_str(val, UNREADABLY));

}

else {

regular\_args = list\_push(regular\_args, val);

}

}

\*args = (\*args)->next;

}

\*args = list\_reverse(regular\_args);

return make\_nil();

}

char\* symbol\_fn(gptr data) {

MalType\* val = data;

return (val->value.mal\_symbol);

}

/\* silence the compiler after swap!, apply, and map

are added to the core \*/

MalType\* apply(MalType\* ast, Env\* env) {

return make\_nil();

}

***types.c***

#include <stdarg.h>

#include <stdio.h>

#include <gc.h>

#include "types.h"

#define ERROR\_BUFFER\_SIZE 128

MalType THE\_TRUE = {MALTYPE\_TRUE, 0, 0, {0}};

MalType THE\_FALSE = {MALTYPE\_FALSE, 0, 0, {0}};

MalType THE\_NIL = {MALTYPE\_NIL, 0, 0, {0}};

inline int is\_sequential(MalType\* val) {

return (val->type == MALTYPE\_LIST || val->type == MALTYPE\_VECTOR);

}

inline int is\_self\_evaluating(MalType\* val) {

return (val->type == MALTYPE\_KEYWORD || val->type == MALTYPE\_INTEGER ||

val->type == MALTYPE\_FLOAT || val->type == MALTYPE\_STRING ||

val->type == MALTYPE\_TRUE || val->type == MALTYPE\_FALSE ||

val->type == MALTYPE\_NIL);

}

inline int is\_list(MalType\* val) {

return (val->type == MALTYPE\_LIST);

}

inline int is\_vector(MalType\* val) {

return (val->type == MALTYPE\_VECTOR);

}

inline int is\_hashmap(MalType\* val) {

return (val->type == MALTYPE\_HASHMAP);

}

inline int is\_nil(MalType\* val) {

return (val->type == MALTYPE\_NIL);

}

inline int is\_string(MalType\* val) {

return (val->type == MALTYPE\_STRING);

}

inline int is\_integer(MalType\* val) {

return (val->type == MALTYPE\_INTEGER);

}

inline int is\_float(MalType\* val) {

return (val->type == MALTYPE\_FLOAT);

}

inline int is\_number(MalType\* val) {

return (val->type == MALTYPE\_INTEGER || val->type == MALTYPE\_FLOAT);

}

inline int is\_true(MalType\* val) {

return (val->type == MALTYPE\_TRUE);

}

inline int is\_false(MalType\* val) {

return (val->type == MALTYPE\_FALSE);

}

inline int is\_symbol(MalType\* val) {

return (val->type == MALTYPE\_SYMBOL);

}

inline int is\_keyword(MalType\* val) {

return (val->type == MALTYPE\_KEYWORD);

}

inline int is\_atom(MalType\* val) {

return (val->type == MALTYPE\_ATOM);

}

inline int is\_error(MalType\* val) {

return (val->type == MALTYPE\_ERROR);

}

inline int is\_callable(MalType\* val) {

return (val->type == MALTYPE\_FUNCTION || val->type == MALTYPE\_CLOSURE);

}

inline int is\_function(MalType\* val) {

return (val->type == MALTYPE\_FUNCTION);

}

inline int is\_closure(MalType\* val) {

return (val->type == MALTYPE\_CLOSURE);

}

inline int is\_macro(MalType\* val) {

return (val->is\_macro);

}

MalType\* make\_symbol(char\* value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_SYMBOL;

mal\_val->value.mal\_symbol = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_integer(long value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_INTEGER;

mal\_val->value.mal\_integer = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_float(double value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_FLOAT;

mal\_val->value.mal\_float = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_keyword(char\* value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_KEYWORD;

mal\_val->value.mal\_keyword = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_string(char\* value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_STRING;

mal\_val->value.mal\_string = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_list(list value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_LIST;

mal\_val->value.mal\_list = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_vector(list value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_VECTOR;

mal\_val->value.mal\_list = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_hashmap(list value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_HASHMAP;

mal\_val->value.mal\_list = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_atom(MalType\* value) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_ATOM;

mal\_val->value.mal\_atom = value;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_function(MalType\*(\*fn)(list args)) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_FUNCTION;

mal\_val->value.mal\_function = fn;

mal\_val->is\_macro = 0;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_closure(Env\* env, MalType\* parameters, MalType\* definition, MalType\* more\_symbol) {

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_CLOSURE;

mal\_val->metadata = NULL;

/\* Allocate memory for embedded struct \*/

MalClosure\* mc = GC\_MALLOC(sizeof(\*mc));

mc->env = env;

mc->parameters = parameters;

mc->definition = definition;

mc->more\_symbol = more\_symbol;

mal\_val->is\_macro = 0;

mal\_val->value.mal\_closure = mc;

return mal\_val;

}

inline MalType\* make\_true() {

return &THE\_TRUE;

}

inline MalType\* make\_false() {

return &THE\_FALSE;

}

inline MalType\* make\_nil() {

return &THE\_NIL;

}

MalType\* make\_error(char\* msg) {

MalType\* mal\_string = GC\_MALLOC(sizeof(\*mal\_string));

mal\_string->type = MALTYPE\_STRING;

mal\_string->value.mal\_string = msg;

MalType\* mal\_val = GC\_MALLOC(sizeof(\*mal\_val));

mal\_val->type = MALTYPE\_ERROR;

mal\_val->value.mal\_error = mal\_string;

mal\_val->metadata = NULL;

return mal\_val;

}

MalType\* make\_error\_fmt(char\* fmt, ...) {

va\_list argptr;

va\_start(argptr, fmt);

char\* buffer = GC\_MALLOC(sizeof(\*buffer) \* ERROR\_BUFFER\_SIZE);

long n = vsnprintf(buffer, ERROR\_BUFFER\_SIZE, fmt, argptr);

va\_end(argptr);

if (n > ERROR\_BUFFER\_SIZE) {

va\_start(argptr, fmt);

buffer = GC\_REALLOC(buffer, sizeof(\*buffer) \* n);

vsnprintf(buffer, n, fmt, argptr);

va\_end(argptr);

}

return make\_error(buffer);

}

MalType\* wrap\_error(MalType\* value) {

MalType\* mal\_error = GC\_MALLOC(sizeof(\*mal\_error));

mal\_error->type = MALTYPE\_ERROR;

mal\_error->metadata = NULL;

mal\_error->value.mal\_error = value;

return mal\_error;

}

MalType\* copy\_type(MalType\* value) {

MalType\* new\_val = GC\_MALLOC(sizeof(\*new\_val));

new\_val->type = value->type;

new\_val->is\_macro = value->is\_macro;

new\_val->value = value->value;

new\_val->metadata = value->metadata;

return new\_val;

}

***step8\_macros.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#include "core.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define SYMBOL\_DO "do"

#define SYMBOL\_IF "if"

#define SYMBOL\_FNSTAR "fn\*"

#define SYMBOL\_QUOTE "quote"

#define SYMBOL\_QUASIQUOTE "quasiquote"

#define SYMBOL\_QUASIQUOTEEXPAND "quasiquoteexpand"

#define SYMBOL\_UNQUOTE "unquote"

#define SYMBOL\_SPLICE\_UNQUOTE "splice-unquote"

#define SYMBOL\_DEFMACROBANG "defmacro!"

#define SYMBOL\_MACROEXPAND "macroexpand"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\*\* env);

void eval\_letstar(MalType\*\* ast, Env\*\* env);

void eval\_if(MalType\*\* ast, Env\*\* env);

MalType\* eval\_fnstar(MalType\* ast, Env\* env);

MalType\* eval\_do(MalType\* ast, Env\* env);

MalType\* eval\_quote(MalType\* ast);

MalType\* eval\_quasiquote(MalType\* ast);

MalType\* eval\_quasiquoteexpand(MalType\* ast);

MalType\* eval\_defmacrobang(MalType\*, Env\*\* env);

MalType\* eval\_macroexpand(MalType\* ast, Env\* env);

MalType\* macroexpand(MalType\* ast, Env\* env);

/\* Use goto to jump here rather than calling eval for tail-call elimination \*/

TCE\_entry\_point:

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* macroexpansion \*/

ast = macroexpand(ast, env);

if (is\_error(ast)) { return ast; }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_letstar(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_IF) == 0) {

/\* TCE - modify ast directly and jump back to eval \*/

eval\_if(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_FNSTAR) == 0) {

return eval\_fnstar(ast, env);

}

else if (strcmp(symbol, SYMBOL\_DO) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

ast = eval\_do(ast, env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUOTE) == 0) {

return eval\_quote(ast);

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTE) == 0) {

ast = eval\_quasiquote(ast);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTEEXPAND) == 0) {

list lst = ast->value.mal\_list;

return eval\_quasiquote(make\_list(lst));

}

else if (strcmp(symbol, SYMBOL\_DEFMACROBANG) == 0) {

return eval\_defmacrobang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_MACROEXPAND) == 0) {

return eval\_macroexpand(ast, env);

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else if (is\_closure(func)) {

MalClosure\* closure = func->value.mal\_closure;

list params = (closure->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(evlst->next);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !closure->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

/\* TCE - modify ast and env directly and jump back to eval \*/

env = env\_make(closure->env, params, evlst->next, closure->more\_symbol);

ast = func->value.mal\_closure->definition;

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

else {

return make\_error\_fmt("first item in list is not callable: '%s'", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

/\* declare as global so it can be accessed by mal\_eval \*/

Env\* global\_env;

MalType\* mal\_eval(list args) {

MalType\* ast = args->data;

return EVAL(ast, global\_env);

}

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

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

global\_env = repl\_env;

ns\* core = ns\_make\_core();

hashmap mappings = core->mappings;

while (mappings) {

char\* symbol = mappings->data;

MalType\*(\*function)(list) = (MalType\*(\*)(list))mappings->next->data;

env\_set\_C\_fn(repl\_env, symbol, function);

/\* pop symbol and function from hashmap/list \*/

mappings = mappings->next->next;

}

env\_set\_C\_fn(repl\_env, "eval", mal\_eval);

/\* add functions written in mal - not using rep as it prints the result \*/

EVAL(READ("(def! not (fn\* (a) (if a false true)))"), repl\_env);

EVAL(READ("(def! load-file (fn\* (f) (eval (read-string (str \"(do \" (slurp f) \"\nnil)\")))))"), repl\_env);

EVAL(READ("(defmacro! cond (fn\* (& xs) (if (> (count xs) 0) (list 'if (first xs) (if (> (count xs) 1) (nth xs 1) (throw \"odd number of forms to cond\")) (cons 'cond (rest (rest xs)))))))"), repl\_env);

/\* make command line arguments available in the environment \*/

list lst = NULL;

for (long i = 2; i < argc; i++) {

lst = list\_push(lst, make\_string(argv[i]));

}

env\_set(repl\_env, make\_symbol("\*ARGV\*"), make\_list(list\_reverse(lst)));

/\* run in script mode if a filename is given \*/

if (argc > 1) {

/\* first argument on command line is filename \*/

char\* load\_command = snprintfbuf(1024, "(load-file \"%s\")", argv[1]);

EVAL(READ(load\_command), repl\_env);

}

/\* run in repl mode when no cmd line args \*/

else {

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.8\n");

puts("Press Ctrl+d to exit\n");

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'def!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'def!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)){

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

void eval\_letstar(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_error("'let\*': missing bindings list");

return;

}

MalType\* bindings = lst->next->data;

MalType\* forms = lst->next->next ? lst->next->next->data : make\_nil();

if (!is\_sequential(bindings)) {

\*ast = make\_error("'let\*': first argument is not list or vector");

return;

}

list bindings\_list = bindings->value.mal\_list;

if (list\_count(bindings\_list) % 2 == 1) {

\*ast = make\_error("'let\*': expected an even number of binding pairs");

return;

}

Env\* letstar\_env = env\_make(\*env, NULL, NULL, NULL);

/\* evaluate the bindings \*/

while(bindings\_list) {

MalType\* symbol = bindings\_list->data;

MalType\* value = EVAL(bindings\_list->next->data, letstar\_env);

/\* early return from error \*/

if (is\_error(value)) {

\*ast = value;

return;

}

env\_set(letstar\_env, symbol, value);

bindings\_list = bindings\_list->next->next;

}

\*env = letstar\_env;

\*ast = forms;

return;

}

void eval\_if(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next || !lst->next->next) {

\*ast = make\_error("'if': too few arguments");

return;

}

if (lst->next->next->next && lst->next->next->next->next) {

\*ast = make\_error("'if': too many arguments");

return;

}

MalType\* condition = EVAL(lst->next->data, \*env);

if (is\_error(condition)) {

\*ast = condition;

return;

}

if (is\_false(condition) || is\_nil(condition)) {

/\* check whether false branch is present \*/

if (lst->next->next->next) {

\*ast = lst->next->next->next->data;

return;

}

else {

\*ast = make\_nil();

return;

}

} else {

\*ast = lst->next->next->data;

return;

}

}

MalType\* eval\_fnstar(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* regularise\_parameters(list\* params, MalType\*\* more);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'fn\*': missing argument list");

}

else if (!lst->next->next) {

return make\_error("'fn\*': missing function body");

}

MalType\* params = lst->next->data;

list params\_list = params->value.mal\_list;

MalType\* more\_symbol = NULL;

MalType\* result = regularise\_parameters(&params\_list, &more\_symbol);

if (is\_error(result)) { return result; }

MalType\* definition = lst->next->next->data;

MalType\* regular\_params = make\_list(params\_list);

return make\_closure(env, regular\_params, definition, more\_symbol);

}

MalType\* eval\_do(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

/\* handle empty 'do' \*/

if (!lst->next) {

return make\_nil();

}

/\* evaluate all but the last form \*/

lst = lst->next;

while (lst->next) {

MalType\* val = EVAL(lst->data, env);

/\* return error early \*/

if (is\_error(val)) {

return val;

}

lst = lst->next;

}

/\* return the last form for TCE evaluation \*/

return lst->data;

}

MalType\* eval\_quote(MalType\* ast) {

list lst = (ast->value.mal\_list)->next;

if (!lst) {

return make\_nil();

}

else if (lst->next) {

return make\_error("'quote': expected exactly one argument");

}

else {

return lst->data;

}

}

MalType\* eval\_quasiquote(MalType\* ast) {

/\* forward reference \*/

MalType\* quasiquote(MalType\* ast);

list lst = ast->value.mal\_list;

/\* no arguments (quasiquote) \*/

if (!lst->next) {

return make\_nil();

}

/\* too many arguments \*/

else if (lst->next->next) {

return make\_error("'quasiquote': expected exactly one argument");

}

return quasiquote(lst->next->data);

}

MalType\* quasiquote(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

MalType\* quasiquote\_vector(MalType\* ast);

/\* argument to quasiquote is self-evaluating: (quasiquote val)

=> val \*/

if (is\_self\_evaluating(ast)) {

return ast;

}

/\* argument to quasiquote is a vector: (quasiquote [first rest]) \*/

else if (is\_vector(ast)) {

return quasiquote\_vector(ast);

}

/\* argument to quasiquote is a list: (quasiquote (first rest)) \*/

else if (is\_list(ast)){

return quasiquote\_list(ast);

}

/\* argument to quasiquote is not self-evaluating and isn't sequential: (quasiquote val)

=> (quote val) \*/

else {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

MalType\* quasiquote\_vector(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

list args = ast->value.mal\_list;

if (args) {

MalType\* first = args->data;

/\* if first element is unquote return quoted \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0) {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

/\* otherwise process like a list \*/

list lst = list\_make(make\_symbol("vec"));

MalType\* result = quasiquote\_list(ast);

if (is\_error(result)) {

return result;

} else {

lst = list\_push(lst, result);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

MalType\* quasiquote\_list(MalType\* ast) {

list args = ast->value.mal\_list;

/\* handle empty list: (quasiquote ())

=> () \*/

if (!args) {

return make\_list(NULL);

}

MalType\* first = args->data;

/\* handle unquote: (quasiquote (unquote second))

=> second \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0 && args->next) {

if (args->next->next) {

return make\_error("'quasiquote': unquote expected exactly one argument");

}

else {

return args->next->data;

}

}

/\* handle splice-unquote: (quasiquote ((splice-unquote first-second) rest))

=> (concat first-second (quasiquote rest)) \*/

else if (is\_list(first) &&

first->value.mal\_list != NULL &&

is\_symbol(first->value.mal\_list->data) &&

strcmp(((MalType\*)first->value.mal\_list->data)->value.mal\_symbol, SYMBOL\_SPLICE\_UNQUOTE) == 0) {

if (!first->value.mal\_list->next) {

return make\_error("'quasiquote': splice-unquote expected exactly one argument");

}

MalType\* first\_second = first->value.mal\_list->next->data;

list lst = list\_make(make\_symbol("concat"));

lst = list\_push(lst, first\_second);

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

}

lst = list\_push(lst, rest);

lst = list\_reverse(lst);

return make\_list(lst);

}

/\* handle all other lists recursively: (quasiquote (first rest))

=> (cons (quasiquote first) (quasiquote rest)) \*/

else {

list lst = list\_make(make\_symbol("cons"));

MalType\* first = quasiquote(args->data);

if (is\_error(first)) {

return first;

} else {

lst = list\_push(lst, first);

}

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

} else {

lst = list\_push(lst, rest);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

}

MalType\* eval\_defmacrobang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'defmacro!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'defmacro!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)) {

result = copy\_type(result);

result->is\_macro = 1;

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

MalType\* eval\_macroexpand(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* macroexpand(MalType\* ast, Env\* env);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_nil();

}

else if (lst->next->next) {

return make\_error("'macroexpand': expected exactly one argument");

}

else {

return macroexpand(lst->next->data, env);

}

}

MalType\* macroexpand(MalType\* ast, Env\* env) {

/\* forward reference \*/

int is\_macro\_call(MalType\* ast, Env\* env);

while(is\_macro\_call(ast, env)) {

list lst = ast->value.mal\_list;

MalType\* macro\_fn = env\_get(env, lst->data);

MalClosure\* cls = macro\_fn->value.mal\_closure;

MalType\* more\_symbol = cls->more\_symbol;

list params\_list = (cls->parameters)->value.mal\_list;

list args\_list = lst->next;

env = env\_make(cls->env, params\_list, args\_list, more\_symbol);

ast = EVAL(cls->definition, env);

}

return ast;

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* regularise\_parameters(list\* args, MalType\*\* more\_symbol) {

/\* forward reference \*/

char\* symbol\_fn(gptr data);

list regular\_args = NULL;

while (\*args) {

MalType\* val = (\*args)->data;

if (!is\_symbol(val)) {

return make\_error\_fmt("non-symbol found in fn argument list '%s'", \

pr\_str(val, UNREADABLY));

}

if (val->value.mal\_symbol[0] == '&') {

/\* & is found but there is no symbol \*/

if (val->value.mal\_symbol[1] == '\0' && !(\*args)->next) {

return make\_error("missing symbol after '&' in argument list");

}

/\* & is found and there is a single symbol after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next &&

is\_symbol((\*args)->next->data) && !(\*args)->next->next)) {

/\* TODO: check symbol is no a duplicate of one already on the list \*/

\*more\_symbol = (\*args)->next->data;

break;

}

/\* & is found and there extra symbols after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next && (\*args)->next->next)) {

return make\_error\_fmt("unexpected symbol after '& %s' in argument list: '%s'", \

pr\_str((\*args)->next->data, UNREADABLY), \

pr\_str((\*args)->next->next->data, UNREADABLY));

}

/\* & is found as part of the symbol and no other symbols \*/

else if (val->value.mal\_symbol[1] != '\0' && !(\*args)->next) {

\*more\_symbol = make\_symbol((val->value.mal\_symbol + 1));

break;

}

/\* & is found as part of the symbol but there are other symbols after \*/

else if (val->value.mal\_symbol[1] != '\0' && (\*args)->next) {

return make\_error\_fmt("unexpected symbol after '%s' in argument list: '%s'", \

pr\_str(val, UNREADABLY), \

pr\_str((\*args)->next->data, UNREADABLY));

}

}

/\* & is not found - add the symbol to the regular argument list \*/

else {

if (list\_findf(regular\_args, val->value.mal\_symbol, symbol\_fn) > 0) {

return make\_error\_fmt("duplicate symbol in argument list: '%s'", pr\_str(val, UNREADABLY));

}

else {

regular\_args = list\_push(regular\_args, val);

}

}

\*args = (\*args)->next;

}

\*args = list\_reverse(regular\_args);

return make\_nil();

}

char\* symbol\_fn(gptr data) {

return (((MalType\*)data)->value.mal\_symbol);

}

/\* used by core functions but not EVAL as doesn't do TCE \*/

MalType\* apply(MalType\* fn, list args) {

if (is\_function(fn)) {

MalType\* (\*fun\_ptr)(list) = fn->value.mal\_function;

return (\*fun\_ptr)(args);

}

else { /\* is\_closure(fn) \*/

MalClosure\* c = fn->value.mal\_closure;

list params = (c->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(args);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !c->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

Env\* env = env\_make(c->env, params, args, c->more\_symbol);

return EVAL(fn->value.mal\_closure->definition, env);

}

}

}

int is\_macro\_call(MalType\* ast, Env\* env) {

/\* not a list \*/

if (!is\_list(ast)) {

return 0;

}

/\* empty list \*/

list lst = ast->value.mal\_list;

if (!lst) {

return 0;

}

/\* first item not a symbol \*/

MalType\* first = lst->data;

if (!is\_symbol(first)) {

return 0;

}

/\* lookup symbol \*/

MalType\* val = env\_get(env, first);

if (is\_error(val)) {

return 0;

}

else {

return (val->is\_macro);

}

}

***step0\_repl.c***

#include <stdio.h>

#include <stdlib.h>

#include <editline/readline.h>

#include <editline/history.h>

#define PROMPT\_STRING "user> "

char\* READ(char\* str) {

return str;

}

char\* EVAL(char\* str) {

return str;

}

void PRINT(char\* str) {

printf("%s\n", str);

}

void rep(char\* str) {

PRINT(EVAL(READ(str)));

}

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

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.1\n");

puts("Press Ctrl+d to exit\n");

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input);

/\* have to release the memory used by readline \*/

free(input);

}

return 0;

}

***step3\_env.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\* env);

MalType\* eval\_letstar(MalType\* ast, Env\* env);

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

return eval\_letstar(ast, env);

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else {

return make\_error\_fmt("Error: first item in list is not callable: %s.", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

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

MalType\* mal\_add(list args);

MalType\* mal\_sub(list args);

MalType\* mal\_mul(list args);

MalType\* mal\_div(list args);

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.3\n");

puts("Press Ctrl+d to exit\n");

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

repl\_env = env\_set\_C\_fn(repl\_env, "+", mal\_add);

repl\_env = env\_set\_C\_fn(repl\_env, "-", mal\_sub);

repl\_env = env\_set\_C\_fn(repl\_env, "\*", mal\_mul);

repl\_env = env\_set\_C\_fn(repl\_env, "/", mal\_div);

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\* env) {

list lst = (ast->value.mal\_list)->next;

/\* TODO: Check the number and types of parameters \*/

MalType\* defbang\_symbol = lst->data;

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, env);

if (!is\_error(result)) {

env\_set(env, defbang\_symbol, result);

}

return result;

}

MalType\* eval\_letstar(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

lst = lst->next;

/\* TODO: Check the bindings list is valid, has an even number of elements, etc\*/

Env\* letstar\_env = env\_make(env, NULL, NULL, NULL);

MalType\* letstar\_bindings = lst->data;

list letstar\_bindings\_list = letstar\_bindings->value.mal\_list;

/\* evaluate the bindings \*/

while(letstar\_bindings\_list) {

MalType\* symbol = letstar\_bindings\_list->data;

MalType\* value = letstar\_bindings\_list->next->data;

letstar\_env = env\_set(letstar\_env, symbol, EVAL(value, letstar\_env));

letstar\_bindings\_list = letstar\_bindings\_list->next->next; /\* pop symbol and value\*/

}

/\* evaluate the forms in the presence of bindings \*/

MalType\* forms = lst->next->data;

return EVAL(forms, letstar\_env);

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

evlst = list\_push(evlst, EVAL(lst->data, env));

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

evlst = list\_push(evlst, EVAL(lst->data, env));

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

evlst = list\_push(evlst, EVAL(lst->data, env));

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* mal\_add(list args) {

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

long sum = 0;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

sum = sum + val->value.mal\_integer;

arg\_list = arg\_list->next;

}

result->value.mal\_integer = sum;

return result;

}

MalType\* mal\_sub(list args) {

long sum;

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

if (arg\_list) {

MalType\* first\_val = arg\_list->data;

arg\_list = arg\_list->next;

/\* TODO: check argument type \*/

sum = first\_val->value.mal\_integer;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

sum = sum - val->value.mal\_integer;

arg\_list = arg\_list->next;

}

}

else {

sum = 0;

}

result->value.mal\_integer = sum;

return result;

}

MalType\* mal\_mul(list args) {

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

long product = 1;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

product \*= val->value.mal\_integer;

arg\_list = arg\_list->next;

}

result->value.mal\_integer = product;

return result;

}

MalType\* mal\_div(list args) {

long product;

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

if (arg\_list) {

MalType\* first\_val = arg\_list->data;

/\* TODO: check argument type \*/

product = first\_val->value.mal\_integer;

arg\_list = arg\_list->next;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

product /= (val->value.mal\_integer);

arg\_list = arg\_list->next;

}

} else {

product = 1;

}

result->value.mal\_integer = product;

return result;

}

***run***

#!/bin/bash

exec $(dirname $0)/${STEP:-stepA\_mal} "${@}"

***step2\_eval.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

/\* evaluate the list \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else {

return make\_error\_fmt("Error: first item in list is not callable: %s.", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

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

MalType\* mal\_add(list args);

MalType\* mal\_sub(list args);

MalType\* mal\_mul(list args);

MalType\* mal\_div(list args);

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.2\n");

puts("Press Ctrl+d to exit\n");

MalType\* func\_add = make\_function(&mal\_add);

MalType\* func\_sub = make\_function(&mal\_sub);

MalType\* func\_mul = make\_function(&mal\_mul);

MalType\* func\_div = make\_function(&mal\_div);

hashmap g = hashmap\_make("+", func\_add);

g = hashmap\_put(g, "-", func\_sub);

g = hashmap\_put(g, "\*", func\_mul);

g = hashmap\_put(g, "/", func\_div);

Env\* repl\_env = GC\_MALLOC(sizeof(\*repl\_env));

repl\_env->data = g;

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = hashmap\_get(env->data, ast->value.mal\_symbol);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

evlst = list\_push(evlst, EVAL(lst->data, env));

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

evlst = list\_push(evlst, EVAL(lst->data, env));

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

evlst = list\_push(evlst, EVAL(lst->data, env));

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* mal\_add(list args) {

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

long sum = 0;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

sum = sum + val->value.mal\_integer;

arg\_list = arg\_list->next;

}

result->value.mal\_integer = sum;

return result;

}

MalType\* mal\_sub(list args) {

long sum;

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

if (arg\_list) {

MalType\* first\_val = arg\_list->data;

arg\_list = arg\_list->next;

/\* TODO: check argument type \*/

sum = first\_val->value.mal\_integer;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

sum = sum - val->value.mal\_integer;

arg\_list = arg\_list->next;

}

}

else {

sum = 0;

}

result->value.mal\_integer = sum;

return result;

}

MalType\* mal\_mul(list args) {

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

long product = 1;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

product \*= val->value.mal\_integer;

arg\_list = arg\_list->next;

}

result->value.mal\_integer = product;

return result;

}

MalType\* mal\_div(list args) {

long product;

MalType\* result = GC\_MALLOC(sizeof(\*result));

result->type = MALTYPE\_INTEGER;

list arg\_list = args;

if (arg\_list) {

MalType\* first\_val = arg\_list->data;

/\* TODO: check argument type \*/

product = first\_val->value.mal\_integer;

arg\_list = arg\_list->next;

while(arg\_list) {

MalType\* val = arg\_list->data;

/\* TODO: check argument type \*/

product /= (val->value.mal\_integer);

arg\_list = arg\_list->next;

}

} else {

product = 1;

}

result->value.mal\_integer = product;

return result;

}

***step1\_read\_print.c***

#include <stdio.h>

#include <stdlib.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* val) {

return val;

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str) {

PRINT(EVAL(READ(str)));

}

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

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.2\n");

puts("Press Ctrl+d to exit\n");

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input);

/\* have to release the memory used by readline \*/

free(input);

}

return 0;

}

***step9\_try.c***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <gc.h>

#include <editline/readline.h>

#include <editline/history.h>

#include "types.h"

#include "reader.h"

#include "printer.h"

#include "env.h"

#include "core.h"

#define SYMBOL\_DEFBANG "def!"

#define SYMBOL\_LETSTAR "let\*"

#define SYMBOL\_DO "do"

#define SYMBOL\_IF "if"

#define SYMBOL\_FNSTAR "fn\*"

#define SYMBOL\_QUOTE "quote"

#define SYMBOL\_QUASIQUOTE "quasiquote"

#define SYMBOL\_QUASIQUOTEEXPAND "quasiquoteexpand"

#define SYMBOL\_UNQUOTE "unquote"

#define SYMBOL\_SPLICE\_UNQUOTE "splice-unquote"

#define SYMBOL\_DEFMACROBANG "defmacro!"

#define SYMBOL\_MACROEXPAND "macroexpand"

#define SYMBOL\_TRYSTAR "try\*"

#define SYMBOL\_CATCHSTAR "catch\*"

#define PROMPT\_STRING "user> "

MalType\* READ(char\* str) {

return read\_str(str);

}

MalType\* EVAL(MalType\* ast, Env\* env) {

/\* forward references \*/

MalType\* eval\_ast(MalType\* ast, Env\* env);

MalType\* eval\_defbang(MalType\* ast, Env\*\* env);

void eval\_letstar(MalType\*\* ast, Env\*\* env);

void eval\_if(MalType\*\* ast, Env\*\* env);

MalType\* eval\_fnstar(MalType\* ast, Env\* env);

MalType\* eval\_do(MalType\* ast, Env\* env);

MalType\* eval\_quote(MalType\* ast);

MalType\* eval\_quasiquote(MalType\* ast);

MalType\* eval\_quasiquoteexpand(MalType\* ast);

MalType\* eval\_defmacrobang(MalType\*, Env\*\* env);

MalType\* eval\_macroexpand(MalType\* ast, Env\* env);

MalType\* macroexpand(MalType\* ast, Env\* env);

void eval\_try(MalType\*\* ast, Env\*\* env);

/\* Use goto to jump here rather than calling eval for tail-call elimination \*/

TCE\_entry\_point:

/\* NULL \*/

if (!ast) { return make\_nil(); }

/\* macroexpansion \*/

ast = macroexpand(ast, env);

if (is\_error(ast)) { return ast; }

/\* not a list \*/

if (!is\_list(ast)) { return eval\_ast(ast, env); }

/\* empty list \*/

if (ast->value.mal\_list == NULL) { return ast; }

/\* list \*/

MalType\* first = (ast->value.mal\_list)->data;

char\* symbol = first->value.mal\_symbol;

if (is\_symbol(first)) {

/\* handle special symbols first \*/

if (strcmp(symbol, SYMBOL\_DEFBANG) == 0) {

return eval\_defbang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_LETSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_letstar(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_IF) == 0) {

/\* TCE - modify ast directly and jump back to eval \*/

eval\_if(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_FNSTAR) == 0) {

return eval\_fnstar(ast, env);

}

else if (strcmp(symbol, SYMBOL\_DO) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

ast = eval\_do(ast, env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUOTE) == 0) {

return eval\_quote(ast);

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTE) == 0) {

ast = eval\_quasiquote(ast);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

else if (strcmp(symbol, SYMBOL\_QUASIQUOTEEXPAND) == 0) {

list lst = ast->value.mal\_list;

return eval\_quasiquote(make\_list(lst));

}

else if (strcmp(symbol, SYMBOL\_DEFMACROBANG) == 0) {

return eval\_defmacrobang(ast, &env);

}

else if (strcmp(symbol, SYMBOL\_MACROEXPAND) == 0) {

return eval\_macroexpand(ast, env);

}

else if (strcmp(symbol, SYMBOL\_TRYSTAR) == 0) {

/\* TCE - modify ast and env directly and jump back to eval \*/

eval\_try(&ast, &env);

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

/\* first element is not a special symbol \*/

MalType\* evaluated\_list = eval\_ast(ast, env);

if (is\_error(evaluated\_list)) { return evaluated\_list; }

/\* apply the first element of the list to the arguments \*/

list evlst = evaluated\_list->value.mal\_list;

MalType\* func = evlst->data;

if (is\_function(func)) {

return (\*func->value.mal\_function)(evlst->next);

}

else if (is\_closure(func)) {

MalClosure\* closure = func->value.mal\_closure;

list params = (closure->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(evlst->next);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !closure->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

/\* TCE - modify ast and env directly and jump back to eval \*/

env = env\_make(closure->env, params, evlst->next, closure->more\_symbol);

ast = func->value.mal\_closure->definition;

if (is\_error(ast)) { return ast; }

goto TCE\_entry\_point;

}

}

else {

return make\_error\_fmt("first item in list is not callable: '%s'", \

pr\_str(func, UNREADABLY));

}

}

void PRINT(MalType\* val) {

char\* output = pr\_str(val, READABLY);

printf("%s\n", output);

}

void rep(char\* str, Env\* env) {

PRINT(EVAL(READ(str), env));

}

/\* declare as global so it can be accessed by mal\_eval \*/

Env\* global\_env;

MalType\* mal\_eval(list args) {

MalType\* ast = args->data;

return EVAL(ast, global\_env);

}

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

Env\* repl\_env = env\_make(NULL, NULL, NULL, NULL);

global\_env = repl\_env;

ns\* core = ns\_make\_core();

hashmap mappings = core->mappings;

while (mappings) {

char\* symbol = mappings->data;

MalType\*(\*function)(list) = (MalType\*(\*)(list))mappings->next->data;

env\_set\_C\_fn(repl\_env, symbol, function);

/\* pop symbol and function from hashmap/list \*/

mappings = mappings->next->next;

}

env\_set\_C\_fn(repl\_env, "eval", mal\_eval);

/\* add functions written in mal - not using rep as it prints the result \*/

EVAL(READ("(def! not (fn\* (a) (if a false true)))"), repl\_env);

EVAL(READ("(def! load-file (fn\* (f) (eval (read-string (str \"(do \" (slurp f) \"\nnil)\")))))"), repl\_env);

EVAL(READ("(defmacro! cond (fn\* (& xs) (if (> (count xs) 0) (list 'if (first xs) (if (> (count xs) 1) (nth xs 1) (throw \"odd number of forms to cond\")) (cons 'cond (rest (rest xs)))))))"), repl\_env);

/\* make command line arguments available in the environment \*/

list lst = NULL;

for (long i = 2; i < argc; i++) {

lst = list\_push(lst, make\_string(argv[i]));

}

env\_set(repl\_env, make\_symbol("\*ARGV\*"), make\_list(list\_reverse(lst)));

/\* run in script mode if a filename is given \*/

if (argc > 1) {

/\* first argument on command line is filename \*/

char\* load\_command = snprintfbuf(1024, "(load-file \"%s\")", argv[1]);

EVAL(READ(load\_command), repl\_env);

}

/\* run in repl mode when no cmd line args \*/

else {

/\* Greeting message \*/

puts("Make-a-lisp version 0.0.9\n");

puts("Press Ctrl+d to exit\n");

while (1) {

/\* print prompt and get input\*/

/\* readline allocates memory for input \*/

char\* input = readline(PROMPT\_STRING);

/\* Check for EOF (Ctrl-D) \*/

if (!input) {

printf("\n");

return 0;

}

/\* add input to history \*/

add\_history(input);

/\* call Read-Eval-Print \*/

rep(input, repl\_env);

/\* have to release the memory used by readline \*/

free(input);

}

}

return 0;

}

MalType\* eval\_ast(MalType\* ast, Env\* env) {

/\* forward references \*/

list evaluate\_list(list lst, Env\* env);

list evaluate\_vector(list lst, Env\* env);

list evaluate\_hashmap(list lst, Env\* env);

if (is\_symbol(ast)) {

MalType\* symbol\_value = env\_get(env, ast);

if (symbol\_value) {

return symbol\_value;

} else {

return make\_error\_fmt("var '%s' not found", pr\_str(ast, UNREADABLY));

}

}

else if (is\_list(ast)) {

list result = evaluate\_list(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_list(result);

} else {

return result->data;

}

}

else if (is\_vector(ast)) {

list result = evaluate\_vector(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_vector(result);

} else {

return result->data;

}

}

else if (is\_hashmap(ast)) {

list result = evaluate\_hashmap(ast->value.mal\_list, env);

if (!result || !is\_error(result->data)) {

return make\_hashmap(result);

} else {

return result->data;

}

}

else {

return ast;

}

}

MalType\* eval\_defbang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'def!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'def!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)){

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

void eval\_letstar(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_error("'let\*': missing bindings list");

return;

}

MalType\* bindings = lst->next->data;

MalType\* forms = lst->next->next ? lst->next->next->data : make\_nil();

if (!is\_sequential(bindings)) {

\*ast = make\_error("'let\*': first argument is not list or vector");

return;

}

list bindings\_list = bindings->value.mal\_list;

if (list\_count(bindings\_list) % 2 == 1) {

\*ast = make\_error("'let\*': expected an even number of binding pairs");

return;

}

Env\* letstar\_env = env\_make(\*env, NULL, NULL, NULL);

/\* evaluate the bindings \*/

while(bindings\_list) {

MalType\* symbol = bindings\_list->data;

MalType\* value = EVAL(bindings\_list->next->data, letstar\_env);

/\* early return from error \*/

if (is\_error(value)) {

\*ast = value;

return;

}

env\_set(letstar\_env, symbol, value);

bindings\_list = bindings\_list->next->next;

}

\*env = letstar\_env;

\*ast = forms;

return;

}

void eval\_if(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next || !lst->next->next) {

\*ast = make\_error("'if': too few arguments");

return;

}

if (lst->next->next->next && lst->next->next->next->next) {

\*ast = make\_error("'if': too many arguments");

return;

}

MalType\* condition = EVAL(lst->next->data, \*env);

if (is\_error(condition)) {

\*ast = condition;

return;

}

if (is\_false(condition) || is\_nil(condition)) {

/\* check whether false branch is present \*/

if (lst->next->next->next) {

\*ast = lst->next->next->next->data;

return;

}

else {

\*ast = make\_nil();

return;

}

} else {

\*ast = lst->next->next->data;

return;

}

}

MalType\* eval\_fnstar(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* regularise\_parameters(list\* params, MalType\*\* more);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_error("'fn\*': missing argument list");

}

else if (!lst->next->next) {

return make\_error("'fn\*': missing function body");

}

MalType\* params = lst->next->data;

list params\_list = params->value.mal\_list;

MalType\* more\_symbol = NULL;

MalType\* result = regularise\_parameters(&params\_list, &more\_symbol);

if (is\_error(result)) { return result; }

MalType\* definition = lst->next->next->data;

MalType\* regular\_params = make\_list(params\_list);

return make\_closure(env, regular\_params, definition, more\_symbol);

}

MalType\* eval\_do(MalType\* ast, Env\* env) {

list lst = ast->value.mal\_list;

/\* handle empty 'do' \*/

if (!lst->next) {

return make\_nil();

}

/\* evaluate all but the last form \*/

lst = lst->next;

while (lst->next) {

MalType\* val = EVAL(lst->data, env);

/\* return error early \*/

if (is\_error(val)) {

return val;

}

lst = lst->next;

}

/\* return the last form for TCE evaluation \*/

return lst->data;

}

MalType\* eval\_quote(MalType\* ast) {

list lst = (ast->value.mal\_list)->next;

if (!lst) {

return make\_nil();

}

else if (lst->next) {

return make\_error("'quote': expected exactly one argument");

}

else {

return lst->data;

}

}

MalType\* eval\_quasiquote(MalType\* ast) {

/\* forward reference \*/

MalType\* quasiquote(MalType\* ast);

list lst = ast->value.mal\_list;

/\* no arguments (quasiquote) \*/

if (!lst->next) {

return make\_nil();

}

/\* too many arguments \*/

else if (lst->next->next) {

return make\_error("'quasiquote': expected exactly one argument");

}

return quasiquote(lst->next->data);

}

MalType\* quasiquote(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

MalType\* quasiquote\_vector(MalType\* ast);

/\* argument to quasiquote is self-evaluating: (quasiquote val)

=> val \*/

if (is\_self\_evaluating(ast)) {

return ast;

}

/\* argument to quasiquote is a vector: (quasiquote [first rest]) \*/

else if (is\_vector(ast)) {

return quasiquote\_vector(ast);

}

/\* argument to quasiquote is a list: (quasiquote (first rest)) \*/

else if (is\_list(ast)){

return quasiquote\_list(ast);

}

/\* argument to quasiquote is not self-evaluating and isn't sequential: (quasiquote val)

=> (quote val) \*/

else {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

MalType\* quasiquote\_vector(MalType\* ast) {

/\* forward references \*/

MalType\* quasiquote\_list(MalType\* ast);

list args = ast->value.mal\_list;

if (args) {

MalType\* first = args->data;

/\* if first element is unquote return quoted \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0) {

list lst = list\_make(ast);

lst = list\_push(lst, make\_symbol("quote"));

return make\_list(lst);

}

}

/\* otherwise process like a list \*/

list lst = list\_make(make\_symbol("vec"));

MalType\* result = quasiquote\_list(ast);

if (is\_error(result)) {

return result;

} else {

lst = list\_push(lst, result);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

MalType\* quasiquote\_list(MalType\* ast) {

list args = ast->value.mal\_list;

/\* handle empty list: (quasiquote ())

=> () \*/

if (!args) {

return make\_list(NULL);

}

MalType\* first = args->data;

/\* handle unquote: (quasiquote (unquote second))

=> second \*/

if (is\_symbol(first) && strcmp(first->value.mal\_symbol, SYMBOL\_UNQUOTE) == 0 && args->next) {

if (args->next->next) {

return make\_error("'quasiquote': unquote expected exactly one argument");

}

else {

return args->next->data;

}

}

/\* handle splice-unquote: (quasiquote ((splice-unquote first-second) rest))

=> (concat first-second (quasiquote rest)) \*/

else if (is\_list(first) &&

first->value.mal\_list != NULL &&

is\_symbol(first->value.mal\_list->data) &&

strcmp(((MalType\*)first->value.mal\_list->data)->value.mal\_symbol, SYMBOL\_SPLICE\_UNQUOTE) == 0) {

if (!first->value.mal\_list->next) {

return make\_error("'quasiquote': splice-unquote expected exactly one argument");

}

MalType\* first\_second = first->value.mal\_list->next->data;

list lst = list\_make(make\_symbol("concat"));

lst = list\_push(lst, first\_second);

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

}

lst = list\_push(lst, rest);

lst = list\_reverse(lst);

return make\_list(lst);

}

/\* handle all other lists recursively: (quasiquote (first rest))

=> (cons (quasiquote first) (quasiquote rest)) \*/

else {

list lst = list\_make(make\_symbol("cons"));

MalType\* first = quasiquote(args->data);

if (is\_error(first)) {

return first;

} else {

lst = list\_push(lst, first);

}

MalType\* rest = quasiquote(make\_list(args->next));

if (is\_error(rest)) {

return rest;

} else {

lst = list\_push(lst, rest);

}

lst = list\_reverse(lst);

return make\_list(lst);

}

}

MalType\* eval\_defmacrobang(MalType\* ast, Env\*\* env) {

list lst = (ast->value.mal\_list)->next;

if (!lst || !lst->next || lst->next->next) {

return make\_error\_fmt("'defmacro!': expected exactly two arguments");

}

MalType\* defbang\_symbol = lst->data;

if (!is\_symbol(defbang\_symbol)) {

return make\_error\_fmt("'defmacro!': expected symbol for first argument");

}

MalType\* defbang\_value = lst->next->data;

MalType\* result = EVAL(defbang\_value, \*env);

if (!is\_error(result)) {

result = copy\_type(result);

result->is\_macro = 1;

\*env = env\_set(\*env, defbang\_symbol, result);

}

return result;

}

MalType\* eval\_macroexpand(MalType\* ast, Env\* env) {

/\* forward reference \*/

MalType\* macroexpand(MalType\* ast, Env\* env);

list lst = ast->value.mal\_list;

if (!lst->next) {

return make\_nil();

}

else if (lst->next->next) {

return make\_error("'macroexpand': expected exactly one argument");

}

else {

return macroexpand(lst->next->data, env);

}

}

MalType\* macroexpand(MalType\* ast, Env\* env) {

/\* forward reference \*/

int is\_macro\_call(MalType\* ast, Env\* env);

while(is\_macro\_call(ast, env)) {

list lst = ast->value.mal\_list;

MalType\* macro\_fn = env\_get(env, lst->data);

MalClosure\* cls = macro\_fn->value.mal\_closure;

MalType\* more\_symbol = cls->more\_symbol;

list params\_list = (cls->parameters)->value.mal\_list;

list args\_list = lst->next;

env = env\_make(cls->env, params\_list, args\_list, more\_symbol);

ast = EVAL(cls->definition, env);

}

return ast;

}

void eval\_try(MalType\*\* ast, Env\*\* env) {

list lst = (\*ast)->value.mal\_list;

if (!lst->next) {

\*ast = make\_nil();

return;

}

if (lst->next->next && lst->next->next->next) {

\*ast = make\_error("'try\*': expected maximum of two arguments");

return;

}

MalType\* try\_clause = lst->next->data;

MalType\* try\_result = EVAL(try\_clause, \*env);

/\* no catch\* clause \*/

if (!is\_error(try\_result) || !lst->next->next) {

\*ast = try\_result;

return;

}

/\* process catch\* clause \*/

MalType\* catch\_clause = lst->next->next->data;

list catch\_list = catch\_clause->value.mal\_list;

if (!catch\_list) {

\*ast = make\_error("'try\*': catch\* clause is empty");

return;

}

MalType\* catch\_symbol = catch\_list->data;

if (strcmp(catch\_symbol->value.mal\_symbol, SYMBOL\_CATCHSTAR) != 0) {

\*ast = make\_error("Error: catch clause is missing catch\* symbol");

return;

}

if (!catch\_list->next || !catch\_list->next->next) {

\*ast = make\_error("Error: catch\* clause expected two arguments");

return;

}

if (!is\_symbol(catch\_list->next->data)) {

\*ast = make\_error("Error: catch\* clause expected a symbol");

return;

}

/\* bind the symbol to the exception \*/

list symbol\_list = list\_make(catch\_list->next->data);

list expr\_list = list\_make(try\_result->value.mal\_error);

/\* TODO: validate symbols and exprs match before calling env\_make \*/

Env\* catch\_env = env\_make(\*env, symbol\_list, expr\_list, NULL);

\*ast = catch\_list->next->next->data;

\*env = catch\_env;

return;

}

list evaluate\_list(list lst, Env\* env) {

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_vector(list lst, Env\* env) {

/\* TODO: implement a real vector \*/

list evlst = NULL;

while (lst) {

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

list evaluate\_hashmap(list lst, Env\* env) {

/\* TODO: implement a real hashmap \*/

list evlst = NULL;

while (lst) {

/\* keys are unevaluated \*/

evlst = list\_push(evlst, lst->data);

lst = lst->next;

/\* values are evaluated \*/

MalType\* val = EVAL(lst->data, env);

if (is\_error(val)) {

return list\_make(val);

}

evlst = list\_push(evlst, val);

lst = lst->next;

}

return list\_reverse(evlst);

}

MalType\* regularise\_parameters(list\* args, MalType\*\* more\_symbol) {

/\* forward reference \*/

char\* symbol\_fn(gptr data);

list regular\_args = NULL;

while (\*args) {

MalType\* val = (\*args)->data;

if (!is\_symbol(val)) {

return make\_error\_fmt("non-symbol found in fn argument list '%s'", \

pr\_str(val, UNREADABLY));

}

if (val->value.mal\_symbol[0] == '&') {

/\* & is found but there is no symbol \*/

if (val->value.mal\_symbol[1] == '\0' && !(\*args)->next) {

return make\_error("missing symbol after '&' in argument list");

}

/\* & is found and there is a single symbol after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next &&

is\_symbol((\*args)->next->data) && !(\*args)->next->next)) {

/\* TODO: check symbol is no a duplicate of one already on the list \*/

\*more\_symbol = (\*args)->next->data;

break;

}

/\* & is found and there extra symbols after \*/

else if ((val->value.mal\_symbol[1] == '\0' && (\*args)->next && (\*args)->next->next)) {

return make\_error\_fmt("unexpected symbol after '& %s' in argument list: '%s'", \

pr\_str((\*args)->next->data, UNREADABLY), \

pr\_str((\*args)->next->next->data, UNREADABLY));

}

/\* & is found as part of the symbol and no other symbols \*/

else if (val->value.mal\_symbol[1] != '\0' && !(\*args)->next) {

\*more\_symbol = make\_symbol((val->value.mal\_symbol + 1));

break;

}

/\* & is found as part of the symbol but there are other symbols after \*/

else if (val->value.mal\_symbol[1] != '\0' && (\*args)->next) {

return make\_error\_fmt("unexpected symbol after '%s' in argument list: '%s'", \

pr\_str(val, UNREADABLY), \

pr\_str((\*args)->next->data, UNREADABLY));

}

}

/\* & is not found - add the symbol to the regular argument list \*/

else {

if (list\_findf(regular\_args, val->value.mal\_symbol, symbol\_fn) > 0) {

return make\_error\_fmt("duplicate symbol in argument list: '%s'", pr\_str(val, UNREADABLY));

}

else {

regular\_args = list\_push(regular\_args, val);

}

}

\*args = (\*args)->next;

}

\*args = list\_reverse(regular\_args);

return make\_nil();

}

char\* symbol\_fn(gptr data) {

return (((MalType\*)data)->value.mal\_symbol);

}

/\* used by core functions but not EVAL as doesn't do TCE \*/

MalType\* apply(MalType\* fn, list args) {

if (is\_function(fn)) {

MalType\* (\*fun\_ptr)(list) = fn->value.mal\_function;

return (\*fun\_ptr)(args);

}

else { /\* is\_closure(fn) \*/

MalClosure\* c = fn->value.mal\_closure;

list params = (c->parameters)->value.mal\_list;

long param\_count = list\_count(params);

long arg\_count = list\_count(args);

if (param\_count > arg\_count) {

return make\_error("too few arguments supplied to function");

}

else if ((param\_count < arg\_count) && !c->more\_symbol) {

return make\_error("too many arguments supplied to function");

}

else {

Env\* env = env\_make(c->env, params, args, c->more\_symbol);

return EVAL(fn->value.mal\_closure->definition, env);

}

}

}

int is\_macro\_call(MalType\* ast, Env\* env) {

/\* not a list \*/

if (!is\_list(ast)) {

return 0;

}

/\* empty list \*/

list lst = ast->value.mal\_list;

if (!lst) {

return 0;

}

/\* first item not a symbol \*/

MalType\* first = lst->data;

if (!is\_symbol(first)) {

return 0;

}

/\* lookup symbol \*/

MalType\* val = env\_get(env, first);

if (is\_error(val)) {

return 0;

}

else {

return (val->is\_macro);

}

}

***stepA\_mal.mal***

;; Testing FFI of "strlen"

(. nil "int32" "strlen" "string" "abcde")

;=>5

(. nil "int32" "strlen" "string" "")

;=>0

;; Testing FFI of "strcmp"

(. nil "int32" "strcmp" "string" "abc" "string" "abcA")

;=>-65

(. nil "int32" "strcmp" "string" "abcA" "string" "abc")

;=>65

(. nil "int32" "strcmp" "string" "abc" "string" "abc")

;=>0

;; Testing FFI of "pow" (libm.so)

(. "libm.so.6" "double" "pow" "double" 2.0 "double" 3.0)

;=>8.000000

(. "libm.so.6" "double" "pow" "double" 3.0 "double" 2.0)

;=>9.000000

***linked\_list.h***

#ifndef \_MAL\_LINKED\_LIST\_H

#define \_MAL\_LINKED\_LIST\_H

/\* simplify references to void pointers \*/

typedef void\* gptr;

/\* linked list is constructed of pairs \*/

typedef struct pair\_s {

gptr data;

struct pair\_s \*next;

} pair;

/\* a list is just a pointer to the pair at the head of the list \*/

typedef pair\* list;

/\* interface \*/

list list\_make(gptr data\_ptr);

list list\_push(list lst, gptr data\_ptr);

gptr list\_peek(list lst);

gptr list\_nth(list lst, int n);

gptr list\_first(list lst);

list list\_rest(list lst);

list list\_pop(list lst);

list list\_reverse(list lst);

long list\_count(list lst);

list list\_concatenate(list lst1, list lst2);

list list\_copy(list lst);

long list\_findf(list lst, char\* keystring, char\*(\*fn)(gptr));

#endif

***linked\_list.c***

#include <stdio.h>

#include <string.h>

#include <gc.h>

#include "linked\_list.h"

list list\_make(gptr data\_ptr) {

return list\_push(NULL, data\_ptr);

}

list list\_push(list lst, gptr data\_ptr) {

pair\* new\_head = GC\_malloc(sizeof(pair));

new\_head->data = data\_ptr;

new\_head->next = lst;

return new\_head;

}

gptr list\_peek(list lst) {

return (lst ? lst->data : NULL);

}

list list\_pop(list lst) {

return (lst ? lst->next : NULL);

}

long list\_count(list lst) {

/\* handle empty case \*/

if (!lst) {

return 0;

}

long counter = 1;

while(lst->next) {

counter++;

lst = lst->next;

}

return counter;

}

list list\_reverse(list lst) {

/\* list is not empty and has more than one element \*/

if (lst && lst->next) {

pair \*prev = NULL, \*next = NULL, \*current = lst;

while (current) {

/\* stash current value of next pointer --> \*/

next = current->next;

/\* reverse the next pointer on current pair <-- \*/

current->next = prev;

/\* move on to next pair and repeat --> \*/

prev = current;

current = next;

}

lst = prev; /\* head of list is in prev when current = NULL \*/

}

return lst;

}

list list\_concatenate(list lst1, list lst2) {

list new\_lst = NULL;

list iterator = NULL;

while (lst2) {

gptr val = lst2->data;

new\_lst = list\_push(new\_lst, val);

lst2 = lst2->next;

}

new\_lst = list\_reverse(new\_lst);

lst1 = list\_reverse(lst1);

iterator = lst1;

while (iterator) {

gptr val = iterator->data;

new\_lst = list\_push(new\_lst, val);

iterator = iterator->next;

}

lst1 = list\_reverse(lst1);

return new\_lst;

}

gptr list\_nth(list lst, int n) {

int idx = 0;

while (lst) {

if (n == idx) {

return lst->data;

}

idx++;

lst = lst->next;

}

return NULL;

}

gptr list\_first(list lst) {

if (lst) {

return lst->data;

}

else {

return NULL;

}

}

list list\_rest(list lst) {

if (lst) {

return lst->next;

}

else {

return NULL;

}

}

list list\_copy(list lst) {

if(!lst) {

return NULL;

}

list new\_lst = NULL;

while(lst) {

new\_lst = list\_push(new\_lst, lst->data);

lst = lst->next;

}

return new\_lst;

}

long list\_findf(list lst, char\* keystring, char\*(\*fn)(gptr)) {

/\* handle empty case \*/

if (!lst) {

return -1;

}

list current = lst;

while(current) {

/\* apply fn to the data to get a string \*/

char\* item = fn(current->data);

if (strcmp(keystring, item) == 0) {

return (current - lst); /\* return the index of the first match \*/

}

else {

current = current->next; /\* skip the next item in the list to\*/

}

}

return -1; /\* not found \*/

}

***hashmap.h***

#ifndef \_MAL\_HASHMAP\_H

#define \_MAL\_HASHMAP\_H

#include "../linked\_list/linked\_list.h"

/\* a hashmap is just a list with alternating key/value pairs \*/

typedef list hashmap;

hashmap hashmap\_make(char\* keystring, gptr data\_ptr);

hashmap hashmap\_put(hashmap map, char\* keystring, gptr data\_ptr);

gptr hashmap\_get(hashmap map, char\* keystring);

gptr hashmap\_getf(hashmap map, char\* keystring, char\*(\*fn)(gptr));

hashmap hashmap\_updatef(hashmap map, char\* keystring, gptr value, char\*(\*fn)(gptr));

#endif

***hashmap.c***

#include <stdio.h>

#include <string.h>

#include <gc.h>

#include "hashmap.h"

hashmap hashmap\_make(char\* keystring, gptr data\_ptr) {

list map = list\_make(data\_ptr);

map = list\_push(map, keystring);

return map;

}

hashmap hashmap\_put(hashmap map, char\* keystring, gptr data\_ptr) {

map = list\_push(map, data\_ptr);

map = list\_push(map, keystring);

return map;

}

gptr hashmap\_get(hashmap map, char\* keystring) {

/\* handle empty case \*/

if (!map) {

return NULL;

}

list lst = map;

while(lst) {

if (strcmp(keystring, (char\*)lst->data) == 0) {

return (lst->next)->data; /\* return next item in list which is the value \*/

}

else {

lst = (lst->next)->next; /\* skip the next item in the list to get to the next key \*/

}

}

return NULL; /\* not found \*/

}

gptr hashmap\_getf(hashmap map, char\* keystring, char\*(\*fn)(gptr)) {

/\* handle empty case \*/

if (!map) {

return NULL;

}

list lst = map;

while(lst) {

/\* apply fn to the data to get a string \*/

char\* item = fn(lst->data);

if (strcmp(keystring, item) == 0) {

return (lst->next)->data; /\* return next item in list which is the value \*/

}

else {

lst = (lst->next)->next; /\* skip the next item in the list to get to the next key \*/

}

}

return NULL; /\* not found \*/

}

hashmap hashmap\_updatef(hashmap map, char\* keystring, gptr value, char\*(\*fn)(gptr)) {

/\* handle empty case \*/

if (!map) {

return NULL;

}

list lst = map;

while(lst) {

/\* apply fn to the data to get a string \*/

char\* item = fn(lst->data);

if (strcmp(keystring, item) == 0) {

(lst->next)->data = value; /\* update the next item in list which is the value \*/

return map; /\* update made \*/

}

else {

lst = (lst->next)->next; /\* skip the next item in the list to get to the next key \*/

}

}

return NULL; /\* no update \*/

}