PLT

Primitive Lisp Technology

Final Report

Michael R. Gargano
mrg2202@columbia.edu

COMS W4115 - Spring 2014

May 14, 2014

Contents

Chapter 1

Introduction

1.1 Language Description

PLT is a functional language in the LISP family of languages and has a minimalistic design philosophy. It is built out of a tiny set of primitive functions with the idea that everything else can be built upon those primitives as needed. While fundamentally, only the concept of lambda is needed to implement all possible computations, writing useful programs in such a language is probably even less productive than writing code in assembly language. Therefore, PLT will not go to that extreme; computer languages are useless without I/O. It has enough primitives to make programming enjoyable even if it does not have as many bells and whistles as Scheme, Common Lisp, or the C family of languages.

1.2 Usage

PLT can be used for general purpose programming. One area where PLT shines is in its ability to handle symbolic processing tasks. Most other languages have some kludgy symbolic processing functionality or force the programmer to use string manipulation for such tasks. I wouldn't suggest writing any major enterprise software in this language but it should be great for quick experimentation and finding solutions to small problems.

1.3 Specifics

PLT works in a similar fashion to other LISPs. All function calls are in prefix notation, e.g. `(+ 2 4)` and everything is an expression that has a value.

Only four data types exist in PLT: integers, floats, symbols, and lists.[1] The only object in the entire language is the S-expression (Symbolic expression). There are no statements or any other kinds of constructs, as mentioned above, everything is an expression that is evaluated. Two different types of S-expressions exist: atoms and lists. The simple rule is anything that's not a list is an atom with one small exception discussed later. The system, like OCaml and Scheme, has no looping construct. Recursive calls are the way to handle iteration in PLT. Since lists are the primary data structure of the language, recursive algorithms usually produce elegant code. Lists are a very versatile data structure. They allow us to have lists, trees, and a primordial type of record structure in our language.

[1] Technically lambdas are also types in the language.

1.4 Primitives

Operation	Explanation
atom?	Is this S-expression an atom or list?
symbol?	Is this S-expression a symbol?
first	Return the first element of a list.
rest	Return the tail of a list.
cons	Add an S-expression to the head of the list.
eq?	Checks to see if two symbolic atoms or structures are equal.
cond	Switch-like conditional expression. Evaluates tests from top to bottom.
quote	Suppress evaluation of an S-expression.
lambda	Define an anonymous function.
define	Bind a value to a symbol.
+	Add numbers, float or integer.
−	Subtract numbers, float or integer.
∗	Multiply numbers, float or integer.
/	Divide numbers, float or integer.
mod	Modulo of two numbers, float or integer.
expt	Exponentiation, float or integer.
=	Compare two numbers (equality), float or integer (suggested integer only).
<	Compare two numbers (less than), float or integer.
>	Compare two numbers (greater than), float or integer.
<=	Compare two numbers (less than or equals), float or integer.
>=	Compare two numbers (greater than or equals), float or integer.
read	Read in an S-expression.
print	Print out an S-expression.
explode	Explode a single symbol into a list of symbols, one for each character.
implode	Condense a list of symbols into a single symbol.

1.5 Examples

Let's start off with some trivial examples to define functions that will be used in our next example.

```
;; define "not", i said this language was minimal :)
(define not
    (lambda (b)
        (cond
            (b nil)    ; if the expression is true, eval to false
```

```
            (t t))))   ; otherwise it's false, eval to true

;; define "and", so minimal
(define and
    (lambda (a b)
        (cond
            (a (cond
                    (b t)    ; a and b are true
                    (t nil)))    ; a is true, b false... false
            (t nil))))   ; a is false, whole thing is false

;; define "or", so very minimal
(define or
    (lambda (a b)
        (cond
            (a t)        ; a is true, it's all true
            (b t)        ; b is true, it's all true
            (t nil))))   ; a and b are false :(

;; check to see if an atom is a number or not
(define number?
    (lambda (n)
        (cond
            ((not (atom? n)) nil)
            ((symbol? n) nil)
            (t t))))

;; get the length of a list, shows recursion
(define length
    (lambda (l)
        (cond
            ((eq? nil l) 0)
            (t (+ 1 (length (rest l)))))))

;; append two lists together
(define append
    (lambda (l1 l2)
        (cond
            ((eq? nil l1) l2)
            (t (cons (first l1) (append (rest l1) l2))))))
```

```
;; get a value from an association list based off a key
(define assoc
    (lambda (key alist)
        (cond
            ((eq? alist nil) nil)  ; list is empty, eval to nil
            ((eq? (first (first alist)) key) (first alist)) ; key is a match
            (t (assoc key (rest alist)))))) ; continue looking for key in rest
```

Below are some of the samples of using these functions.

```
(and t nil) → nil
(or t nil) → t
(and t t) → t
(not t) → nil
(number?  2) → t
(number?  (quote k)) → nil
(length (quote (a b c d))) → 4
(length (quote (a))) → 1
(append (quote (a b c)) (quote (d e f))) → (a b c d e f)
(append (quote (a b)) nil) → (a b)
(assoc (quote k1) (quote ((k1 .  2) (k2 .  5)))) → (k1 .  2)
(assoc (quote k3) (quote ((k1 .  2) (k2 .  5) (k3 .  3)))) → (k3
.  3)
(assoc (quote k4) (quote ((k1 .  2) (k2 .  5)))) → nil
```

The next example is a symbolic differentiation example adapted from the example in SICP[1]. It's a little long, but shows how much can be built up from the primitives to perform some computer algebra.

```
;; is the first atom a "+" sign
(define sum?
    (lambda (exp)
        (eq? (first exp) (quote +))))

;; is the first atom a "*" sign
(define prod?
    (lambda (exp)
```

```
        (eq? (first exp) (quote *))))

;; is the first atom "expt"
(define expt?
    (lambda (exp)
        (eq? (first exp) (quote expt))))

;; ex.  (+ 1 2) -> 1
(define first-term
    (lambda (exp)
        (first (rest (exp)))))  ; get second item in list

;; ex. (+ 1 2) -> 2
(define second-term
    (lambda (exp)
        (first (rest (rest (exp))))))    ; get third item in list

;; create a simplified sum expression with the two given expressions
(define create-sum
    (lambda (exp1 exp2)
        (cond
            ; both exp1 and exp2 can just be added together
            ((and (number? exp1) (number? exp2)) (+ exp1 exp2)) ; sum the digits
            ((and (number? exp1) (= exp1 0)) exp2) ; if adding with zero
            ((and (number? exp2) (= exp2 0)) exp1)
            ; otherwise do the usual
            (t (cons (quote +) (cons exp1 (cons exp2 nil)))))))

;; create a simplified prod expression with the two given expressions
(define create-prod
    (lambda (exp1 exp2)
        (cond
            ; both exp1 and exp2 can just be multiplied together
            ((and (number? exp1) (number? exp2)) (* exp1 exp2)) ; do the prod
            ; if one expression is a number and that number is 0, eval to 0
            ((or
                (and (number? exp1) (= exp1 0))
                (and (number? exp2) (= exp2 0))) 0)
            ; if one expr is a number and that number is 1, eval to the other expr
            ((and (number? exp1) (= exp1 1)) exp2)
```

```
            ((and (number? exp2) (= exp2 1)) exp1)
            ; otherwise do the usual
            (t (cons (quote *) (cons exp1 (cons exp2 nil)))))))

;; negate expression
(define negate
    (lambda (exp)
        (cond
            ((number? exp) (- 0 exp))
            (t (create-prod -1 exp)))))

;; create a simplified difference expression with the two given expressions
(define create-diff
    (lambda (exp1 exp2)
        (cond
            ((and (number? exp1) (number? exp2)) (- exp1 exp2))
            ((and (number? exp1) (= exp1 0)) (negate exp2))
            ((and (number? exp2) (= exp2 0)) exp1)
            (t (cons (quote -) (cons exp1 (cons exp2 nil)))))))

;; create a expt expression
(define create-expt
    (lambda (base expt)
        (cond
            ((and (number? base) (number? expt)) (expt base expt))
            (t (cons (quote expt) (cons base (cons expt nil)))))))

;; the main show, call this with a quoted symbolic expression and the variable
;;  with which the derivative is to be taken with respect to
(define derivative
    (lambda (exp var)
        (cond
            ; it's a constant, dc/dx = 0
            ((number? exp) 0)
            ; it's a variable
            ((symbol? exp)
                (cond
                    ((eq? exp var) 1)    ; if it's with respect to var
                    (t 0)))              ; otherwise the deriv. is 0
            ; it's a sum of two expressions
```

```
((sum? exp)        ; take derivative of each term
                   ;  and leave sum
    (create-sum
        (derivative (first-term exp) var)
        (derivative (second-term exp) var)))
; it's a product of two expressions
((prod? exp)
    (create-sum
        (create-prod
            (derivative (first-term exp) var)
            (second-term exp))
        (create-prod
            (first-term exp)
            (derivative (second-term exp) var))))
; it's an exponentiation
((expt? exp)
    (create-prod
        (create-prod
            (second-term exp)
            (create-expt
                (first-term exp)
                (create-diff (second-term exp) 1)))
        (derivative (first-term exp))))
(t (print (quote "error cannot take derivative"))))))
```

So, in about 100 lines of code, we've created a small computer algebra system that can take derivatives of some polynomial equations. Below are some samples of calling the derivative function shown above.

`(derivative (quote 2) (quote x))` → `0`
`(derivative (quote x) (quote x))` → `1`
`(derivative (quote (+ x 27)) (quote x))` → `1`
`(derivative (quote (* x y)) (quote x))` → `y`
`(derivative (quote (* x y)) (quote y))` → `x`
`(derivative (quote (+ x (* 2 x))) (quote x))` → `3`
`(derivative (quote (expt x 4)) (quote x))` → `(* 4 (expt x 3))`
`(derivative (quote (* (* x y) (+ x 27))) (quote x))` → `(+ (* y (+ x 27)) (* x y))`

1.6 Other Things To Note

Some additional things we can learn about PLT from the examples is that `nil` is equivalent to `()` and they both represent falsehood. They are both atoms and lists in the language. The symbol `t` evaluates to itself, like `nil`, and can be used to represent truth. Symbols can be contained in double quotes (`"`), they are still symbols but can have spaces and other characters normally not permitted in other Lisps. PLT is a lexically scoped Lisp-1.[2] The language also contains dotted pairs. When creating a list, each cell has a pointer to a value and a pointer to the next item in the list until the last item in the list points to `nil` as the next list item. This cell structure can be used to point to two values instead of another item in this list, when this happens the structure is represented as a dotted pair `(a . b)`. Finally, comments begin with `;` and terminate at the end of the line.

[2] A LISP designated as a Lisp-1, means there is only one namespace for variables and function names.

Chapter 2

Language Tutorial

This is a quick and dirty introduction to some of the features of the PLT language.

2.1 Basics

Everything is evaluated in PLT.

2.1.1 Numbers

Integers and doubles evaluate to themselves.
`2` → `2`
`2.7` → `2.7`

2.1.2 Symbols

Symbols are usually bound to values. Their associated value is looked up and returned Define a binding like this:
`(define a 5)` → `5`
this binds `a` to 5 and returns 5 as the value now,
`a` → 5
`t` and `nil` are special symbols, they evaluate to themselves.
`t` → `t`
`nil` → `nil`
To surpress evaluation of a symbol, we use the `quote` special form.
`(quote a)` → `a`

2.1.3 Lists

A list is a group of space separated S-Expressions between parentheses.
`(quote (a b c))` → `(a b c)`
You will see why we need these in a moment.
Lists can be pulled apart and put back together.
`(first (quote (4 5 6)))` → `4`
`first` returns the first element of a list.
`(rest (quote (4 5 6)))` → `(5 6)`
`rest` returns everything but the first element of the list, the "rest" of it.
Things can be easily added to the head of a list like so:
`(cons (quote y) (quote (9 x 7 3.4 w)))` → `(y 9 x 7 3.4 w)`
`cons` takes an object and a list and constructs a new list with the object at its head and the previous list in the tail.

2.1.4 Functions

Functions are called by evaluating expressions like this:
`(+ 4 5)` → `9`
It is called in prefix notation. The first element of the list is the function to call, the arguments are evaluated left-to-right and then passed in. By not quoting the previous list show in the last section, PLT would try to lookup a function called `a` and evaluate `b` and `c` before calling it on them.
You can create your own functions in the following manner:
`(lambda (x) (+ x x))` → `<lambda:  0x83859283>`
even anonymous functions evaluate to something... a lambda.
Anonymous functions can be called like so:
`((lambda (x) (+ x x)) 3)` → `6`
here the function is in prefix notation and is called on the argument 3, returning the result 6. We can save such functions for later use by binding them to a symbol.
`(define double (lambda (x) (+ x x)))` → `<lambda:  0x385daf43>`
This can now be called and used, just like the built-in functions.
`(double 3)` → `6`
A list and explanation of useful built-in procedures can be found in the language reference manual.
PLT has no looping constructs, instead recursion is used. Here is an example program to compute a factorial using recursion:

```
(define factorial
  (lambda (n)
```

```
  (cond
((= 0 n) 1)
(t (* n (factorial (- n 1)))))))
```

We can see a few new things here. Functions can call themselves, a new structure called `cond` (see the next section), and three new built-in functions `=`, `*`, and `-`. `=` preforms a numerical comparison and returns true or false. The other two functions are self explanatory.

2.1.5 Conditionals

`cond` takes the form

```
(cond
 (condition1 result1)
 (condition2 result2) . . . )
```

The first condition that evaluates to true, has its result returned, if no conditions match `nil` is returned. We've learned something else new too. `t` and `nil` can be use for true and false, respectively. Actually, anything that's not `nil` is true. Now we can make complete sense of the `factorial` example above.

2.2 Advanced

2.2.1 Closures

Look at this function definition and the following code:

```
(define make-counter
  (lambda () (define count 0)
    (lambda () (define count (+ count 1))
               count)))

(define counter1 (make-counter))
```

The first function definition defines a function, `make-counter`, that returns another function as a result. Now this can be done with PLT, the thing that makes this a little more special is that the variable `count`, which is defined by `make-counter`, is no longer lexically accessible to the anonymous function when it returns, therefore it must make a reference to that outer value for later use. Now when we call `(counter1)` we get sequential numbers as expected on each call. But we can still use `make-counter` to create as many new counters as we wish.

Chapter 3

Language Manual

3.1 Introduction

This document will describe the the PLT language. Both its syntax and semantics will be expounded upon. It is intended to be the definitive reference for the language.

3.2 Lexical Conventions

A program is contained entirely in a single file. This file is then processed as a stream of tokens and is evaluated from the topmost S-Expression to the bottom.

3.2.1 Tokens

There a handful of tokens in PLT: symbols, integers, doubles, and delimiters for dotted pairs and lists (a special type of dotted pair). *White space* is considered to be spaces, tabs, newlines, and carriage returns. It is used to delineate tokens, but is otherwise ignored.

3.2.2 Comments

Comments begin with the the semicolon character (`;`) and continue until the end of the line. The semicolon character cannot occur inside of unquoted symbols, otherwise from where it appears, to the end of the line will be considered a comment. Double quoted symbols can contain the semicolon character.

3.2.3 Keywords

PLT does not really contain any keywords. There are many built-in special forms and functions for ease of use, but the symbols those functions are bound to does not make them any more special than any other symbol in the system. They are just processed a little differently than user defined functions and variables.[1] The one symbol that *is* treated specially is the symbol `nil`. `nil` is a symbol, but is also considered to be an empty list. This duality makes it special and it is actually a specific token recognized by the compiler.

3.2.4 Integers

An integer is a sequence of decimal digits that may optionally begin with a sign (`+` or `-`). All integers are signed (whether the sign is specified or not) and is equivalent in size and range to OCaml's `int` and the corresponding rules for 32-bit and 64-bit machines.

3.2.5 Doubles

Doubles have an integer part, a decimal point, a fraction part, an exponent part (denoted by `e` or `E`) containing an optionally signed integer. The integer, fraction, and exponent parts contain a sequence of decimal digits. The integer part or fraction part of the the double may be omitted, but not both. Either the decimal point or the exponent part of the double may be omitted, but not both. This IEEE double-precision floating point number is equivalent to OCaml's `float` and C's `double` in size and range.

3.2.6 Symbols

Symbols in PLT can take two different forms, a doubly quoted symbol or a regular unquoted symbol.

Unquoted

Symbols in this form must start with an alpha character (lower or upper) or any of the following characters: `‘ ’ ~ ! @ # $ % ^ & * + = : ? / < > . , - _ + { } [ ] | \` followed by any combination and number (including zero) of the preceding characters mentioned and/or digits (`0 - 9`).

[1] `define`, `cond`, and `lambda` are not bound to function definitions, they are special forms and are handled separately inside the compiler.

One exception is the symbol consisting of only a single dot (`.`). While this is allowed by the above definition, the single dot is used in the definition of dotted pairs and must be ether used in combination with other characters or as part of a double quoted symbol.

Double Quoted

Symbols in this form start with a double quote character (`"`) and end with one as well. Any character inside of the double quotes are allowed except for another double quote character. This gives the user a lot of flexibility in terms of naming, but care must be taken as `"27"` and `"-2.36e+248"` are valid double quoted symbols. In PLT these double quoted symbols take on almost a dual role as strings and can be used as such.

3.3 Syntax Notation

Most of the syntax for PLT has been mostly defined above (believe it or not). Only two other syntactic structures remain to be defined: dotted pairs and lists.

3.3.1 Dotted Pairs

The fundamental underlying structure in PLT is the dotted pair, sometimes called a cons cell or ordered pair. The dotted pair structure has two pointers that can point to symbols or another dotted pair. The dotted pair has the following structure:
`(` *first S-Expression pointed to* `.` *second S-Expression pointed to* `)`
So, after its namesake, we can see a dotted pair is a pointer to two other S-Expressions with a dot, surrounded by whitespace[2], separating the two of them.

3.3.2 Lists

Lists, at the end of the day, are really just a special case in the possible structures of dotted pairs. A list is just dotted pairs arranged in the form of a singly-linked list. A list has the following structure:
`(` *element1 element2 element3* `)`
This can be equiviallently exapanded into the following dotted list structure:
`(` *element1* `.` `(` *element2* `.` `(` *element3* `.` `nil` `)))`

[2]See the *Symbols* subsection to understand why.

The singly-linked list show in the structure above, like a lists, is `nil` terminated.

3.4 Objects

PLT only contains one type of object, the S-Expression (Symbolic Expression). There are two types of S-Expressions: atoms and dotted pairs. Atoms consist of the following: symbols, integers, and doubles. Dotted pairs, as we saw before, can contain atoms and/or other dotted pairs.

3.4.1 Forms

A form is any object that is meant to be evaluated: integers, doubles, symbols, compound forms. PLT programs consist of lists of forms, evaluating one after the other.

3.4.2 Self Evaluating Forms

These include integers, doubles, and `nil`. These objects all evaluate to themselves.

3.4.3 Variables

Symbols can be used to name variables. When a symbol is evaluated as a form, the value of the bound variable is the result.
Local variables are lexically scoped in PLT.

3.4.4 Compound Forms

A non-empty list that is a function form, special form, or lambda form.

3.4.5 Function Forms

A non-empty list where the first element is the name of a function that is meant to be called on the arguments (what is obtained after individually evaluating the remaining elements in the list).
`(` *func-name arg1 arg2 ...* `)`

3.4.6 Special Forms

A form with special evaluation rules. Examples in PLT include: `cond`, `quote`, `define`, and `lambda`. It is a list where the first element is the special operator. The other items in the list may or may not be evaluated as they in the other forms.
`(` *special-op arg1 arg2 ...* `)`

Cond

`cond` the test conditions sequenced in a `cond` form may or may not all be evaluated. The condition *test*s are evaluated in order until one of them returns a non-`nil` result. Only then will that conditions *consequent*s be evaluated in sequential order, leaving all other conditions *consequent*s unevaluated as well as the remaining tests. If no *test* return true, none of the *consequent*s are evaluated.

Quote

`quote` simply suppresses evaluation of its argument and returns it unevaluated.

Define

`define` suppresses the evaluation of its *id* argument, using it as if it were a quoted symbol. Its *form* is then evaluated and bound to the *id*.

Lambda

`lambda` does not evaluate its *arg-list* or *body*. The *arg-list* defines parameter symbols that are to be bound into the `lambda`'s environment when it is called. Only at this time is the *body* evaluated.

3.4.7 Lambda Forms

A non-empty list where the first element is a lambda function meant to be applied to the following arguments (what is obtained after individually evaluating the remaining elements in the list).
`(` `(` `lambda` `(` *arg-id1 arg-id2 ...* `)` *body...* `)` *arg1 arg2 ...* `)`

3.5 Predicates

Predicates answer true/false questions. The value for true in PLT is the symbol `t`[3], which evaluates to itself. The value for false is `nil`.

Function	Description
`atom?` *sexpr*	Is the S-Expression an atom?
`symbol?` *sexpr*	Is the S-Expression a symbol?
`eq?` *sexpr sexpr*	Are two symbolic atoms or two structures equal?

3.6 Control Structure

Special Operator	Description
`cond (`*test consequent ...*`) ...`	Evaluates test, if it evaluates to true, performs the consequents returning the last value as the result, otherwise it moves to the next test. If no tests succeed, the value of `cond` is `nil`
`define` *id form*	Bind the value of the evaluated form to the identifier.
`quote` *sexpr*	Simply returns the unevaluated form as is. Usually used for defining data.
`lambda (`*arg-id ...*`)` *body*	Create a function that takes the arguments in the arg list and executes the body (which can be a sequence of expressions) when called. Returns the last value in the body as the result.

In PLT there are no explicit control structures for looping. Looping is done through recursion. Recursively defined functions can provide equivalent functionality to looping.

3.7 Numbers

3.7.1 Comparisons

Predicate	Condition
`=` *arg1 arg2*	Are the two numbers the same.
`<` *arg1 arg2*	First number less than the second.
`>` *arg1 arg2*	First number is greater than the second.
`<=` *arg1 arg2*	First number is less than or equal to the second.
`>=` *arg1 arg2*	First number is greater than or equal to the second.

*All work with integers and doubles. It is suggested that `=` only be used with integers.

[3]Any value that is not `nil` can be used to indicate truth, `t` is a convenience symbol.

3.7.2 Arithmetic Operations

Function	Description
`+` *arg ...*	Sum all of the arguments together. Add the argument to 0 if only one is given.
`-` *arg ...*	Successively subtract the arguments from the first. Subtract the argument from 0 if only one is given.
`*` *args ...*	Multiply all of the arguments together.
`/` *arg ...*	Successively divide the first argument by the following arguments. If only one argument is supplied, its inverse is returned.
`mod` *num div*	Modulus of number and divisor.

*All work with integers and doubles.

3.7.3 Exponential and Logarithmic Functions

Function	Description
`expt` *base pow*	Raises the first number to the power of the second.
`log` *num base*	Log of the first number in the base of the second.

3.8 List/Dotted Pair Operations

Function	Description
`cons` *sexpr dp*	Add an S-Expression to the head of a list. Create new dotted pair with the first pointer pointing to sexpr and the second pointer pointing to dp.
`first` *dp*	Return the first element of the list or first pointer in the dotted pair.
`rest` *dp*	Return the tail of the list of the second pointer in the dotted pair.

3.9 Input/Output

Function	Description
`read`	Read in an S-Expression. Only very well formatted ones. Very delicate.
`print` *sexpr*	Print out an S-Expression.

3.10 Symbol Manipulation

Function	Description
`explode` *symbol*	Explode a single symbol into a list of symbols, one for each character in the original symbol.
`implode` *list*	Condense a list of symbols into a single symbol.

Chapter 4

Project Plan

4.1 Team Responsibilities

As this is a team of one, all project related responsibilities fall to me. They include: analysis, design, implementation, testing, and documentation.

4.2 Program Style

4.2.1 Introduction

Again as a team of one, this was not as essential as it would be in a team environment. Below are some rules of thumb I use and my thoughts on coding style that guided me.

4.2.2 General Principles

I think code should be mostly self documenting. This is achieved by picking good variable and function names when possible. Also, you mostly want to write idiomatic code in whatever language you are using, be it LISP, C, or OCaml. This makes reading the code all that easier for people familiar with the language. The caveat here, however, is to not write idiomatic code when it makes the code's logic too confusing. For example you can write some idiomatic things in C that could take experienced C programers minutes or hours to determine what the code is doing. This is not very good style in my opinion. Having said that, in my C code, over the years I've adopted quite a few of the conventions for formatting used in the Linux code base, though not all. I'm not usually hard and fast with coding conventions, because sometimes, I think some exceptions to your formatting can make things

clearer and more ascetically appealing. On the OCaml side of things, since I don't have much experience with OCaml it is hard to write idiomatic code. I did my best at nesting things where it made things clearer. I actually cheated a little here, with OCaml, by running the code through a pretty printer to keep it nice and clean. It made the code about 200 lines longer, but it's much cleaner to read.

4.2.3 Tabs

All tabs are expanded to spaces in the source files. It is too difficult to work with actual tabs across different editors.

4.2.4 Comments

I like to comment things, actually, over-comment. The thought is the more information the better. Especially when looking at the code a few years from now. In general, however, I try to make the code be as self-commenting as possible by trying to pick good names and writing clear, sometimes non-idiomatic code. Again, however, I wish I had more time to comment even more.

4.3 Project Timeline and Log

Here is the list of significant project milestones and deliverables. The timeline was completely influenced by the project deliverables, noted with by (*). The main implementation work on the compiler and runtime library was highly iterative and spread across the entire date range from start to completion. There were no well defined milestones along the way of that portion. If there really were any they can be thought of as: implement garbage collector, implement runtime framework, write compiler to target the runtime framework, and implement built-in library functions. As mentioned already, however, these did not have very clean boundaries and were done mostly in parallel and iterated over to refine them as things moved along.

1-21-2014	Started thinking about and designing language.
1-31-2014	Language mostly defined.
2-01-2014	Project white paper delivered.*
3-10-2014	Scanner completed.
3-12-2014	Parser completed.
3-13-2014	Language reference manual due.*
4-23-2014	Compiler and Runtime library work started.
5-14-2014	Project and final report due.*

4.4 Software Development Environment

All work was performed on Apple Machintoshes on the OS X operating system. The compiler is written completely in OCaml and emits C code, which was then compiled using the `clang` C compiler. Apple's XCode IDE was used for development and debugging of the runtime library, which is written entirely in standard C. It was also used to debug PLT generated programs and inspect them for memory leaks and other issues.[1] The UNIX shell, and its scripting language, was used to automate some of the testing of the compiler.

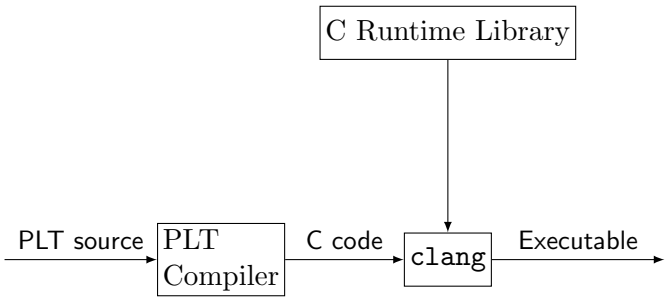
[1] We don't want our garbage collector, or anything else really, leaking memory.

Chapter 5

Architectural Design

5.1 Introduction

The high level architecture consists of the PLT compiler (written in OCaml), and the C runtime library. A PLT source program is feed to the compiler, which generates and emits the program as C, the C file is then taken and compiled against the C runtime library with `clang`, the output of this phase is an executable.



5.2 PLT Compiler

The PLT compiler's architecture is very straight forward. It consists of a scanner, a parser, and a code generator. The generator emits intermediate C code that is then consumed by and linked with the C runtime library.

The scanner was built using the `ocamllex` tool. The parser was built using the `ocamlyacc` tool. The code generation phase is written entirely in OCaml and walks the AST emitting C code while preforming some semantic checking along the way. The scanner, parser, and code generator all indicate errors by throwing exceptions and aborting. If I had the time, I would try to make this process more informative to the user. The scanner and parser are both auto-generated, so I can't say much more about the internal structures used there. The code generator walks the AST recursively, generating C code into a string buffer as it goes. Sometimes an intermediate buffer is created to generate code for sub-expressions that must be evaluated before the code we are generating can be. That result is then incorporated into the main string buffer in the correct order. The only other data structure is really just a very simple symbol table. Since, PLT is a highly dynamic language, our symbol table does not need to keep track of type information and such. The symbol table consists solely of a set on a stack. As we enter and exit lexical scopes, new sets are created and pushed on the stack to retain the symbol information for that new scope and popped off on exit of scope. The stack is searched from top to bottom to determine if a symbol is in scope and new symbols are added to the set at the top. This gives us the ability to determine if something is in scope or not and whether or not it needs to be handled specially when creating lexical closures.

File	Description
main.ml	The file that kicks off the scanner, followed by parser, and code generator. It also contains a quick and dirty pretty printer, used for some debugging.
ast.mli	Types used for the AST.
scanner.mll	The scanner defined in `ocamllex` format.
parser.mly	The parser defined in `ocamlyacc` format.
compiler.ml	The main deal. The code generator in straight OCaml.

5.3 C Runtime Library

The C runtime library consists of a mark-and-sweep garbage collector, environment stack structure and routines for handling them, dynamic type structures and routines for handling them, a way to dynamically call lambdas, and finally the built-in primitive functions that can be used in PLT. Many of the structures in the runtime library are linked lists. Lots of efficiency can be gained if the structures were transformed into something more appropriate for the problem, but time did not permit that. One example in

particular is the use of linked lists in the environment stack. Much improvement in speed is to be gained here if a hash table were to be used instead.

File	Description
builtin.(c|h)	The code that implements the built-in functions.
environment.(c|h)	The code that defines the environment structures and utility functions.
error.(c|h)	Global error function.
functions.(c|h)	Function to call lambdas dynamically.
init.(c|h)	Function to call to initialize program, defines globals and built-ins in environment.
memory.(c|h)	Garbage collector structures and functions for allocating memory and mark-and-sweep.
structs.(c|h)	Defines structures for the dynamic type and utility functions for allocating them.

The main structure is the `TAGGED_VALUE`, it defines a dynamic type that can take on the role of either a symbol, integer, double, lambda, or cons cell. Types are determined at runtime. The environment, `ENV`, is essentially the call stack of the program. It stores `BINDING`s that associate a symbol with a value in a linked list. When a `TAGGED_VALUE` is allocated, a reference to it is stored in a `MEM_REF`, which in turn is stored in a linked list by the garbage collector. This allows the garbage collector to keep track of everything that has been allocated. Things are marked for saving if they can be reached through the environment, otherwise, the memory references are traversed and if the object isn't to be saved, it is freed. All of this runtime code is rather large and involved, and hopeful won't be counted against the developer during grading time.

Chapter 6

Test Plan

6.1 Introduction

As mentioned previously, testing was done in a mostly iterative fashion. Sample input, programs, and code snippets were being tested along the whole of the compiler development.

6.2 Scanner and Parser Testing

When the scanner and parser were being developed, small files of test input data were used along with a pretty printer (that could annotate the emitted AST) to determine if the scanner and parser were behaving as expected on various inputs. Even with this coverage up front, an incorrect case was found during the later development stages that required a change to the parser.

6.3 Code Generation and Runtime Library

During the very initial stages of development for the code generator, code snippets were entered by hand and the generated source examined to determine if the code was being correctly generated. After that, once the main runtime functionality had been implemented, sample programs were written to exercise some of the runtime functionally. This again was a manual process as it involved compiling and debugging code through the XCode IDE and looking at memory usage and other statistics through profiling tools to determine if the garbage collector and runtime system were performing as expected. Once things were working end-to-end, the automated testing system was set up to run through sample coverage test cases. Unfortunately,

just working through the issues in the runtime did not afford enough time to build a comprehensive test suite and most of the testing was conducted manually. Again, as mentioned before, this was highly iterative, although I wouldn't go so far as to call it Agile.

6.4 Automation

Automation was preformed using a modified shell script based off the one shown in class. It ran the source through the OCaml compiler, compiled the generated C code against the runtime library and compared the source output against reference output.

6.5 Test Suite

Here is an example of one of the test suite programs used. It was used to test the functionality of the built-in functions. Other representative test programs are below, each testing something different.

```
;test quote and S-expression parsing
(print (quote "Quote and S-Expressions"))
(print (quote a))
(print (quote abba))
(print (quote "Hello you!"))
(print (quote 27.7))
(print (quote -27.7))
(print (quote -27e10))
(print (quote -27e-5))
(print (quote 27e10))
(print (quote 27e-5))
(print (quote 27.7e10))
(print (quote 27.7e-5))
(print (quote 1))
(print (quote -1))
(print (quote nil))
(print (quote t))
(print (quote ()))
(print (quote (a b)))
(print (quote (a b c d)))
(print (quote (a . b)))
```

```
(print (quote (a b c . d)))
(print (quote ((a) . (b))))
(print (quote ((a b) . (c d))))
(print (quote (a b c . nil)))
(print (quote (a b c . ())))
(print (quote (a . nil)))
(print (quote (a . ())))
(print (quote (nil . a)))
(print (quote (() . a)))
(print (quote (() . nil)))
(print (quote (() . ())))
(print (quote (((a b c . nil) . d))))
(print (quote (((a b c . nil) . (a b . c)))))

; test comments
; test implode and explode
(print (quote "Implode and Explode"))
(define a (explode (quote "Hello World!!!")))
(print a)
(define b (explode (quote this_is_a_pretty_big_symbol)))
(print b)
(define c (implode a))
(print c)
(define d (implode b))
(print d)
(print (implode (quote ("-" "hey you" " out there " "on " "the " "road" "."))))
(print (explode (implode (quote ("-" "hey you" " out there " "on " "the " "road" ".")

;test atom
(print (quote "Atom?"))
(print (atom? (quote a)))
(print (atom? (quote "hi there")))
(print (atom? 27))
(print (atom? 27.000))
(print (atom? t))
(print (atom? nil))
(print (atom? (quote (a b c))))
(print (atom? (quote (a . b))))

;test symbol
```

```
(print (quote "Symbol?"))
(print (symbol? (quote a)))
(print (symbol? (quote "hi there")))
(print (symbol? 27))
(print (symbol? 27.000))
(print (symbol? t))
(print (symbol? nil))
(print (symbol? (quote (a b c))))
(print (symbol? (quote (a . b))))

;test eqp
(print (quote "eq?"))
(print (eq? 1 1))
(print (eq? 0 0))
(print (eq? -1 -1))
(print (eq? 1 1.0))
(print (eq? 1 2))
(print (eq? 1 0))
(print (eq? 0.0 0.00))
(print (eq? -1 1))
(print (eq? 2.0 2.00))
(print (eq? 2.0 2.001))
(print (eq? -2.0 -2.0))
(print (eq? (quote a) (quote a)))
(print (eq? (quote a) (quote A)))
(print (eq? (quote a) (quote b)))
(print (eq? (quote abba) (quote abba)))
(print (eq? (quote aBba) (quote abba)))
(print (eq? (quote "hey you there!!!") (quote "hey you there!!!")))
(print (eq? (quote "hey you therE!!!") (quote "hey you there!!!")))
(print (eq? (quote (a . b)) (quote (a . b))))
(print (eq? 1 (quote a)))
(print (eq? 1.0 (quote a)))
(print (eq? (quote (a b c)) (quote (a b c))))
(print (eq? t t))
(print (eq? (quote t) (quote t)))
(print (eq? (quote t) t))
(print (eq? nil nil))
(print (eq? (quote nil) (quote nil)))
(print (eq? (quote nil) nil))
```

```
(print (eq? (quote (a b c . d)) (quote (a b c . d))))
(print (eq? (quote ((a b c) . (d e))) (quote ((a b c) . (d e)))))
(print (eq? (quote ((a b c) . (f e))) (quote ((a b c) . (d e)))))
(print (eq? (quote (((a b) c) . (d e))) (quote (((a b) c) . (d e)))))
(print (eq? (quote (((a b) c) . (d e))) (quote (((a f) c) . (d e)))))
(print (eq? (quote (((a . b) c) . (d e))) (quote (((a b) c) . (d e)))))
(print (eq? (quote (((a . b) c) . (d e))) (quote (((a . b) c) . (d e)))))
(print (eq? (quote (((a 1) c) . (d e))) (quote (((a b) c) . (d e)))))
(print (eq? (quote (((a 1) c) . (d e))) (quote (((a 1.0) c) . (d e)))))
(print (eq? (quote (((a 1.0) c) . (d e))) (quote (((a 1.0) c) . (d e)))))
(define l1 (lambda () 7))
(define l2 (lambda () 7))
(print (eq? l1 l1))
(print (eq? l2 l2))
(print (eq? l1 l2))
(print (eq? (quote (((a l1) c) . (d e))) (quote (((a l1) c) . (d e)))))
(print (eq? (quote (((a l1) c) . (d e))) (quote (((a l2) c) . (d e)))))
(print (eq? (quote (a b c)) (quote (a b c . nil))))
(print (eq? (quote (a nil b)) (quote (a nil b))))
(print (eq? (quote (a)) 1))
(print (eq? (quote (a . b)) 1))

;test equals
(print (quote "="))
(print(= 0 0))
(print(= 1 1))
(print(= -1 -1))
(print(= 1.0 1.0))
(print(= -1.0 -1.0))

;test less than
(print (quote "<"))
(print(< 0 0))
(print(< 1 2))
(print(< 2 1))
(print(< -1 2))
(print(< 2 -1))
(print(< .5 1))
(print(< 1 .5))
(print(< -.5 1))
```

```
(print(< 1 -.5))

;test greater than
(print (quote ">"))
(print(> 0 0))
(print(> 1 2))
(print(> 2 1))
(print(> -1 2))
(print(> 2 -1))
(print(> .5 1))
(print(> 1 .5))
(print(> -.5 1))
(print(> 1 -.5))

;test less than or equals
(print (quote "<="))
(print(<= 0 0))
(print(<= 1 1))
(print(<= .5 .5))
(print(<= -1 -1))
(print(<= -.5 -.5))
(print(<= 1 2))
(print(<= 2 1))
(print(<= -1 2))
(print(<= 2 -1))
(print(<= .5 1))
(print(<= 1 .5))
(print(<= -.5 1))
(print(<= 1 -.5))

;test greater than of equals
(print (quote ">="))
(print(>= 0 0))
(print(>= 1 1))
(print(>= .5 .5))
(print(>= -1 -1))
(print(>= -.5 -.5))
(print(>= 1 2))
(print(>= 2 1))
(print(>= -1 2))
```

```
(print(>= 2 -1))
(print(>= .5 1))
(print(>= 1 .5))
(print(>= -.5 1))
(print(>= 1 -.5))

;test addition
(print (quote "+"))
(print(+ 2))
(print (+ 0))
(print (+ 0.0))
(print(+ 2.0))
(print(+ -2.0))
(print(+ -1))
(print(+ 2 3))
(print(+ 2 -1))
(print(+ 2 -1.0))
(print(+ 2.0 -1))
(print(+ 2.0 -1.0))
(print(+ 2 3 4 5))
(print(+ 2 2.0))
(print(+ 2 -2.0))
(print(+ -2 2.0))
(print(+ 2 2.0 3.0 4 5.5))
(print(+ 2.0 3.0 4.0 5.5))

;test subtraction
(print (quote "-"))
(print (- 2))
(print (- -2))
(print (- 2.0))
(print (- -2.9))
(print (- 2 5))
(print (- 5 2))
(print (- 5 -2))
(print (- -5 2))
(print (- -5 -2))
(print (- -5.0 2))
(print (- 5.0 -2))
(print (- -5.0 -2))
```

```
(print (- 5.0 2))
(print (- 2 5.0))
(print (- 2 3 5 9 0 10))
(print (- 10 1 2 3))
(print (- 10.0 2 1.0 3))
(print (- 10.0 2.5 2.6 1.9))
(print (- 5.0 3.0 2.0 4.0))

;test multiplication
(print (quote "*"))
(print(* 2))
(print(* 2.0))
(print(* -2.0))
(print(* -1))
(print(* 2 3))
(print(* 2 -1))
(print(* 2 -1.0))
(print(* 2.0 -1))
(print(* 2.0 -1.0))
(print(* 2 3 4 5))
(print(* 2 2.0))
(print(* 2 -2.0))
(print(* -2 2.0))
(print(* 2 2.0 3.0 4 5.5))
(print(* 2.0 3.0 4.0 5.5))

;test division
(print (quote "/"))
(print (/ 1))
(print (/ 2))
(print (/ 5))
(print (/ -1))
(print (/ -2))
(print (/ -5))
(print (/ 1 1))
(print (/ 1 5))
(print (/ 5 1))
(print (/ 20 11))
(print (/ 20 10))
(print (/ 1.0))
```

```
(print (/ -1.0))
(print (/ -2.0))
(print (/ -.5))
(print (/ 90 10 3 3))
(print (/ 90 -10 3 3))
(print (/ 90 -10 -3 3))
(print (/ 90 10.0 3.0))
(print (/ 90 10 2 1))
(print (/ 90.0 10.0 3.0 3.0))
(print (/ 90.0 -10.0 3.0 3.0))
(print (/ 90.0 -10.0 -3.0 3.0))
(print (/ 90.0 10 3.0 3.0))
(print (/ 90.0 -10 3.0 3.0))
(print (/ 90.0 10 -3.0 3.0))
(print (/ 90.0 10 3.0 -3))

;test mod
(print (quote "mod"))
(print (mod 4 2))
(print (mod 5 2))
(print (mod -4 2))
(print (mod -5 2))
(print (mod 4 -2))
(print (mod 5 -2))
(print (mod -4 -2))
(print (mod -5 -2))
(print (mod 4.0 2.0))
(print (mod 5.0 2.0))
(print (mod -4.0 2.0))
(print (mod -5.0 2.0))
(print (mod 4.0 -2.0))
(print (mod 5.0 -2.0))
(print (mod -4.0 -2.0))
(print (mod -5.0 -2.0))

;test expt
(print (quote "expt"))
(print (expt 2 2))
(print (expt 3 3))
(print (expt 3 -1))
```

```
(print (expt -3 1))
(print (expt 55 0))

;test log
(print (quote "log"))
(print (log 4 2))
(print (log 3 3))
(print (log 27 3))

;list processing
;test cons
(print (quote "cons"))
(print (cons (quote a) (quote b)))
(print (cons (quote a) (quote ())))
(print (cons (quote a) (quote nil)))
(print (cons (quote a) ()))
(print (cons (quote a) nil))
(print (cons (quote (a b)) nil))
(print (cons (quote (a)) (quote (b c))))
(print (cons (quote a) (quote (b c))))
(print (cons (quote (a b)) (quote c)))
(print (cons (quote (a (b c))) (quote (d))))

;test first
(print (quote "first"))
(print (first (quote ())))
(print (first (quote nil)))
(print (first ()))
(print (first nil))
(print (first (quote (a . b))))
(print (first (quote (a b c))))
(print (first (quote ((a b) c d))))
(print (first (quote ((a b) c . d))))
(print (first (quote ((a b) c d . e))))
(print (first (quote (1 b c))))

;test rest
(print (quote "rest"))
(print (rest (quote ())))
(print (rest (quote nil)))
```

```
(print (rest ()))
(print (rest nil))
(print (rest (quote (a . b))))
(print (rest (quote (a . 1))))
(print (rest (quote (a b c))))
(print (rest (quote ((a b) c d))))
(print (rest (quote ((a b) c . d))))
(print (rest (quote ((a b) c d . e))))
(print (rest (quote ((a b) c . (d e)))))
(print (rest (quote (1 b c))))

; finished
(print (quote "Done"))
```

6.6 Representative Programs

6.6.1 PLT Program 1 (Basic Test)

```
(define func (lambda (x y) (+ x y)))
(define a (func 2 4))
(define b (func 4 2))
(print (eq? a b))
```

6.6.2 Generated Source Program 1 (Basic Test)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "init.h"
#include "memory.h"
#include "builtin.h"
#include "structs.h"
#include "functions.h"
#include "environment.h"


extern ENV *global_env;
extern TAGGED_VALUE *t;
```

```
extern TAGGED_VALUE *nil;

TAGGED_VALUE *___lambda_num_0(TAGGED_VALUE*);


TAGGED_VALUE *___lambda_num_0(TAGGED_VALUE *this)
{
TAGGED_VALUE *___eval_num_0 = lookup_symbol(alloc_symbol("x"),this);
TAGGED_VALUE *___eval_num_1 = lookup_symbol(alloc_symbol("y"),this);
TAGGED_VALUE *___eval_num_2 = call_lambda(lookup_symbol(alloc_symbol("+"),this),2,___
return ___eval_num_2;
}


int main(int argc, char* argv[])
{
TAGGED_VALUE *this = NULL; /* no enclosing lambda, at top level */

init(); /* create primordial environment */

TAGGED_VALUE *___eval_num_3 = alloc_lambda(___lambda_num_0,2,alloc_params(2,"x","y"),
TAGGED_VALUE *___eval_num_4 = add_binding(alloc_symbol("func"),___eval_num_3,this);
gc();

TAGGED_VALUE *___eval_num_5 = alloc_int(2);
TAGGED_VALUE *___eval_num_6 = alloc_int(4);
TAGGED_VALUE *___eval_num_7 = call_lambda(lookup_symbol(alloc_symbol("func"),this),2,
TAGGED_VALUE *___eval_num_8 = add_binding(alloc_symbol("a"),___eval_num_7,this);
gc();

TAGGED_VALUE *___eval_num_9 = alloc_int(4);
TAGGED_VALUE *___eval_num_10 = alloc_int(2);
TAGGED_VALUE *___eval_num_11 = call_lambda(lookup_symbol(alloc_symbol("func"),this),2
TAGGED_VALUE *___eval_num_12 = add_binding(alloc_symbol("b"),___eval_num_11,this);
gc();

TAGGED_VALUE *___eval_num_13 = lookup_symbol(alloc_symbol("a"),this);
TAGGED_VALUE *___eval_num_14 = lookup_symbol(alloc_symbol("b"),this);
TAGGED_VALUE *___eval_num_15 = call_lambda(lookup_symbol(alloc_symbol("eq?"),this),2,
TAGGED_VALUE *___eval_num_16 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
```

```
gc();


gc_on_exit(); /* clean up all objects, no marky... just sweepy */
clear_global_env(); /* clean up any bindings left in the global env with refs to nowh


return 0;
}
```

6.6.3 PLT Program 2 (Test Closures)

```
(define make-counter
    (lambda (name)
        (define count 0)
        (lambda ()
            (define count (+ count 1))
            (cons name (cons count nil)))))
(define a (make-counter (quote a)))
(define b (make-counter (quote b)))
(print (a))
(print (a))
(print (a))
(print (a))
(print (b))
(print (b))
(print (b))
(print (b))
(print (a))
(print (b))
(print (b))
```

6.6.4 Generated Code Program 2 (Test Closures)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "init.h"
```

```
#include "memory.h"
#include "builtin.h"
#include "structs.h"
#include "functions.h"
#include "environment.h"


extern ENV *global_env;
extern TAGGED_VALUE *t;
extern TAGGED_VALUE *nil;

TAGGED_VALUE *___lambda_num_1(TAGGED_VALUE*);
TAGGED_VALUE *___lambda_num_0(TAGGED_VALUE*);


TAGGED_VALUE *___lambda_num_1(TAGGED_VALUE *this)
{
TAGGED_VALUE *___eval_num_2 = lookup_symbol(alloc_symbol("count"),this);
TAGGED_VALUE *___eval_num_3 = alloc_int(1);
TAGGED_VALUE *___eval_num_4 = call_lambda(lookup_symbol(alloc_symbol("+"),this),2,___
TAGGED_VALUE *___eval_num_5 = add_binding(alloc_symbol("count"),___eval_num_4,this);
TAGGED_VALUE *___eval_num_6 = lookup_symbol(alloc_symbol("name"),this);
TAGGED_VALUE *___eval_num_7 = lookup_symbol(alloc_symbol("count"),this);
TAGGED_VALUE *___eval_num_8 = nil;
TAGGED_VALUE *___eval_num_9 = call_lambda(lookup_symbol(alloc_symbol("cons"),this),2,
TAGGED_VALUE *___eval_num_10 = call_lambda(lookup_symbol(alloc_symbol("cons"),this),2
return ___eval_num_10;
}
TAGGED_VALUE *___lambda_num_0(TAGGED_VALUE *this)
{
TAGGED_VALUE *___eval_num_0 = alloc_int(0);
TAGGED_VALUE *___eval_num_1 = add_binding(alloc_symbol("count"),___eval_num_0,this);
TAGGED_VALUE *___eval_num_11 = alloc_lambda(___lambda_num_1,0,alloc_params(0),NULL,cr
return ___eval_num_11;
}


int main(int argc, char* argv[])
{
TAGGED_VALUE *this = NULL; /* no enclosing lambda, at top level */
```

```
init(); /* create primordial environment */

TAGGED_VALUE *___eval_num_12 = alloc_lambda(___lambda_num_0,1,alloc_params(1,"name"),
TAGGED_VALUE *___eval_num_13 = add_binding(alloc_symbol("make-counter"),___eval_num_1
gc();

TAGGED_VALUE *___eval_num_14 = alloc_symbol("a");
TAGGED_VALUE *___eval_num_15 = call_lambda(lookup_symbol(alloc_symbol("make-counter")
TAGGED_VALUE *___eval_num_16 = add_binding(alloc_symbol("a"),___eval_num_15,this);
gc();

TAGGED_VALUE *___eval_num_17 = alloc_symbol("b");
TAGGED_VALUE *___eval_num_18 = call_lambda(lookup_symbol(alloc_symbol("make-counter")
TAGGED_VALUE *___eval_num_19 = add_binding(alloc_symbol("b"),___eval_num_18,this);
gc();

TAGGED_VALUE *___eval_num_20 = call_lambda(lookup_symbol(alloc_symbol("a"),this),0);
TAGGED_VALUE *___eval_num_21 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_22 = call_lambda(lookup_symbol(alloc_symbol("a"),this),0);
TAGGED_VALUE *___eval_num_23 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_24 = call_lambda(lookup_symbol(alloc_symbol("a"),this),0);
TAGGED_VALUE *___eval_num_25 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_26 = call_lambda(lookup_symbol(alloc_symbol("a"),this),0);
TAGGED_VALUE *___eval_num_27 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_28 = call_lambda(lookup_symbol(alloc_symbol("b"),this),0);
TAGGED_VALUE *___eval_num_29 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_30 = call_lambda(lookup_symbol(alloc_symbol("b"),this),0);
TAGGED_VALUE *___eval_num_31 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();
```

```
TAGGED_VALUE *___eval_num_32 = call_lambda(lookup_symbol(alloc_symbol("b"),this),0);
TAGGED_VALUE *___eval_num_33 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_34 = call_lambda(lookup_symbol(alloc_symbol("b"),this),0);
TAGGED_VALUE *___eval_num_35 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_36 = call_lambda(lookup_symbol(alloc_symbol("a"),this),0);
TAGGED_VALUE *___eval_num_37 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_38 = call_lambda(lookup_symbol(alloc_symbol("b"),this),0);
TAGGED_VALUE *___eval_num_39 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();

TAGGED_VALUE *___eval_num_40 = call_lambda(lookup_symbol(alloc_symbol("b"),this),0);
TAGGED_VALUE *___eval_num_41 = call_lambda(lookup_symbol(alloc_symbol("print"),this),
gc();


gc_on_exit(); /* clean up all objects, no marky... just sweepy */
clear_global_env(); /* clean up any bindings left in the global env with refs to nowh


return 0;
}
```

6.6.5 PLT Program 3 (Test Recursion)

```
(define count-down (lambda (x)
    (cond
        ((= x 0) (print x))
        (t (print x) (count-down (- x 1))))))
(print (count-down 200))
```

6.6.6 Generated Code Program 3 (Test Recursion)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "init.h"
#include "memory.h"
#include "builtin.h"
#include "structs.h"
#include "functions.h"
#include "environment.h"


extern ENV *global_env;
extern TAGGED_VALUE *t;
extern TAGGED_VALUE *nil;

TAGGED_VALUE *___lambda_num_0(TAGGED_VALUE*);


TAGGED_VALUE *___lambda_num_0(TAGGED_VALUE *this)
{
    TAGGED_VALUE *___cond_result_num_0 = nil;
    TAGGED_VALUE *___eval_num_0 = lookup_symbol(alloc_symbol("x"),this);
    TAGGED_VALUE *___eval_num_1 = alloc_int(0);
    TAGGED_VALUE *___eval_num_2 = call_lambda(lookup_symbol(alloc_symbol("="),this),2
    if(nil != ___eval_num_2)
    {
        TAGGED_VALUE *___eval_num_3 = lookup_symbol(alloc_symbol("x"),this);
        TAGGED_VALUE *___eval_num_4 = call_lambda(lookup_symbol(alloc_symbol("print")
        ___cond_result_num_0 = ___eval_num_4;
        goto COND_END_NUM_0;
    }
    TAGGED_VALUE *___eval_num_5 = lookup_symbol(alloc_symbol("t"),this);
    if(nil != ___eval_num_5)
    {
        TAGGED_VALUE *___eval_num_6 = lookup_symbol(alloc_symbol("x"),this);
        TAGGED_VALUE *___eval_num_7 = call_lambda(lookup_symbol(alloc_symbol("print")
        TAGGED_VALUE *___eval_num_8 = lookup_symbol(alloc_symbol("x"),this);
        TAGGED_VALUE *___eval_num_9 = alloc_int(1);
```

```
        TAGGED_VALUE *___eval_num_10 = call_lambda(lookup_symbol(alloc_symbol("-"),th
        TAGGED_VALUE *___eval_num_11 = call_lambda(lookup_symbol(alloc_symbol("count-
        ___cond_result_num_0 = ___eval_num_11;
        goto COND_END_NUM_0;
    }
COND_END_NUM_0:
    return ___cond_result_num_0;
}


int main(int argc, char* argv[])
{
    TAGGED_VALUE *this = NULL; /* no enclosing lambda, at top level */

    init(); /* create primordial environment */

    TAGGED_VALUE *___eval_num_12 = alloc_lambda(___lambda_num_0,1,alloc_params(1,"x")
    TAGGED_VALUE *___eval_num_13 = add_binding(alloc_symbol("count-down"),___eval_num
    gc();

    TAGGED_VALUE *___eval_num_14 = alloc_int(200);
    TAGGED_VALUE *___eval_num_15 = call_lambda(lookup_symbol(alloc_symbol("count-down
    gc();


    gc_on_exit(); /* clean up all objects, no marky... just sweepy */
    clear_global_env(); /* clean up any bindings left in the global env with refs to


    return 0;
}
```

Chapter 7

Lessons Learned

I know it's been said before, but time is not your friend on this project. It is literally the fire I which I am burning. I started thinking about this project right from the start, but still feel an enormous time crunch. As many of the details that you think you have covered, I guarantee you there are at least twice as many things that you haven't thought of. You many think you know how something works, but you may realize when it comes to implementation time that you don't or there are many edge cases that you have missed and need to understand. I think that part of the reason this is so hard for this project is because, for most of us, you are designing a new language in a bubble. There is not necessarily a need for the language, and the language design is not really iterative, it is just one deliverable. I think that when most companies, or other people design languages, they have lots of time to sit with what they've done and experiment and realize where there are holes or how things can be done more elegantly. That's why things like Java, C, C++, and OCaml don't see the light of day for years or, now, have version numbers. We don't have time for that. My advice is to come up with as many sample programs for this invented language as quickly as possible when designing it. This will let you see its pain-points early and help to smooth out it's edges for a cleaner design. Also think very hard about the architecture of the compiler and get to work on any runtime you might have as quickly as possible. I know it's hard because you're learning it as you go, but it will make things easier on you.

Chapter 8

Appendix

8.1 Compiler

8.1.1 Makefile

Makefile for OCaml source.

```
TARFILES = Makefile scanner.mll parser.mly ast.mli
compiler.ml main.ml

OBJS = parser.cmo scanner.cmo compiler.cmo main.cmo

plt : $(OBJS)
     ocamlc -g -o plt $(OBJS)

scanner.ml : scanner.mll
     ocamllex scanner.mll

parser.ml parser.mli : parser.mly
     ocamlyacc parser.mly

%.cmo : %.ml
     ocamlc -c $<

%.cmi : %.mli
     ocamlc -c $<

plt.tar.gz : $(TARFILES)
     cd .. && tar zcf plttar/plt.tar.gz $(TARFILES:%=plt/%)

.PHONY : clean
clean :
     rm -f plt parser.ml parser.mli scanner.ml *.cmo *.cmi

# Generated by ocamldep *.ml *.mli
compiler.cmo : ast.cmi
compiler.cmx : ast.cmi
main.cmo : scanner.cmo parser.cmi compiler.cmo ast.cmi
main.cmx : scanner.cmx parser.cmx compiler.cmx ast.cmi
parser.cmo : ast.cmi parser.cmi
parser.cmx : ast.cmi parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
ast.cmi :
parser.cmi : ast.cmi
```

8.1.2 ast.mli

AST types.

```
(* three types of atoms (nil is really a symbol, but handled specially) *)
type atom = | Symbol of string | Int of int | Double of float | Nil

(* S-expressions *)
type sexpr = | Atom of atom | DottedPair of sexpr * sexpr

(* the root *)
type prog = | Prog of sexpr list
```

8.1.3 scanner.mll

Scanner definition.

```
{ open Parser
  exception SyntaxError of string  (* define a syntax error exception *)
}

(* predefined regexes *)
(* int and float stuff *)
let digit = ['0'-'9']
let digits = ['0'-'9']+
let sign = ['-''+']
let e = ['e''E']sign?
(* symbol stuff *)
let lalpha = ['a'-'z']
let ualpha = ['A'-'Z']
let oalpha =
['`''~''!''@''#''$''%''^''&''*''+''=''<''>''.'',''-''_''{''}''[']''|''\
\''\'']
let symbol_start = (lalpha|ualpha|oalpha)

(* token parsing *)
rule token =
parse [' ' '\t' '\r' '\n'] { token lexbuf }
| ';' { comment lexbuf }  (* comments *)
| '(' { LPAREN }
| ')' { RPAREN }
| '.' { DOT }
| "nil" { NIL }
(* double matching *)
| (sign)?(digits)e(digits) as double { DOUBLE(float_of_string double) }
| (sign)?(digit*)'.'(digits)(e(digits))? as double { DOUBLE(float_of_string
```

```
double) }
| (sign)?(digits)'.'(digit*)(e(digits))? as double { DOUBLE(float_of_string
double) }
(* integer matching *)
| (sign)?(digits) as integer { INTEGER(int_of_string integer) }
(* symbol matching *)
| '"' { let buf = Buffer.create 27 in
          Buffer.add_char buf '"';
          read_dbl_quoted_symbol buf lexbuf }
| symbol_start+(symbol_start|digit)* as symbol { SYMBOL(symbol) }
| eof { EOF }
| _ { raise (SyntaxError("Syntax error: " ^ Lexing.lexeme lexbuf)) }

(* ignore comments until end of line or file *)
and comment =
parse ['\n' '\r'] {token lexbuf}
| _ { comment lexbuf}
| eof { EOF }

(* different state for double quoted symbols *)
and read_dbl_quoted_symbol buf =
parse '"' { Buffer.add_char buf '"';
            SYMBOL(Buffer.contents buf) }
| [^'"'] { Buffer.add_string buf (Lexing.lexeme lexbuf);
           read_dbl_quoted_symbol buf lexbuf }
```

8.1.4 parser.mly

Parser definition.

```
%{ open Ast  %}

%token LPAREN RPAREN DOT NIL EOF
%token <int> INTEGER
%token <float> DOUBLE
%token <string> SYMBOL

%start prog
%type <Ast.prog> prog

%%

prog:
  sexprs EOF           { Prog(List.rev $1) }

sexpr:
  atom                 { Atom($1) }
| LPAREN RPAREN        { Atom(Nil) }
| LPAREN sexprs RPAREN {  let rec buildList = function
                            [] -> Atom(Nil)
                            |h::t ->  DottedPair(h, buildList t)
                          in buildList (List.rev $2) }
| LPAREN sexprs DOT sexpr RPAREN { let rec buildDottedPair = function
                                     [] -> $4
                                     |h::t ->  DottedPair(h, buildDottedPair t)
                                   in buildDottedPair (List.rev $2) }

sexprs:
  sexprs sexpr         { $2::$1 }
```

```
| sexpr                 { [$1] }

atom:
  SYMBOL                { Symbol($1) }
| INTEGER               { Int($1) }
| DOUBLE                { Double($1) }
| NIL                   { Nil }
```

8.1.5 compiler.ml

The code generator.

```
open Ast

module StringMap = Map.Make(String)

module StringSet = Set.Make(String)

(* Info for functions, the name they map to in C and the min,max arity *)
type symbol_info = | Function of string * int * int

(* set of special form names *)
let special_forms =
  List.fold_left (fun s k -> StringSet.add k s) StringSet.empty
    [ "cond"; "quote"; "define"; "lambda" ]

(* set of the built-in functions, and some of their related information, such as
    what they map to in C and their acceptable arities *)
let builtin_functions =
  List.fold_left (fun m (k, v) -> StringMap.add k v m) StringMap.empty
    [ ("atom?", (Function ("atomp", 1, 1)));
      ("symbol?", (Function ("symbolp", 1, 1)));
      ("eq?", (Function ("eqp", 2, 2))); ("=", (Function ("equal", 2, 2)));
      ("<", (Function ("lessthan", 2, 2)));
      (">", (Function ("greaterthan", 2, 2)));
      ("<=", (Function ("lessthanequal", 2, 2)));
      (">=", (Function ("greaterthanequal", 2, 2)));
      ("+", (Function ("add", 1, 536870911)));
      ("-", (Function ("sub", 1, 536870911)));
      ("*", (Function ("prod", 1, 536870911)));
      ("/", (Function ("divide", 1, 536870911)));
```

```
      ("mod", (Function ("mod", 2, 2))); ("expt", (Function ("expt", 2, 2)));
      ("log", (Function ("log_num_base", 2, 2)));
      ("cons", (Function ("cons", 2, 2)));
      ("first", (Function ("first", 1, 1)));
      ("rest", (Function ("rest", 1, 1)));
      ("read", (Function ("read_inp", 0, 0)));
      ("print", (Function ("print", 1, 1)));
      ("explode", (Function ("explode", 1, 1)));
      ("implode", (Function ("implode", 1, 536870911))) ]

(* stack of symbol tables, one per scope *)
let sym_stack = Stack.create ()

(* add global symbol table, just need symbols, types are all dynamic *)
let create_new_scope () = Stack.push StringSet.empty sym_stack

(* pop a scope from the stack *)
let remove_scope () = Stack.pop sym_stack

(* get the current scope *)
let get_current_scope () = Stack.top sym_stack

(* is the symbol defined, in the top scope only *)
let sym_exists_in_current_scope sym =
  StringSet.mem sym (get_current_scope ())

(* is the symbol defined in anything that is scope accessible, helper function,
    keeps second temp stack, pops off from sym stack and then puts everything back
*)
```

```
let rec lookup_sym sym temp_stack =
  if StringSet.mem sym (get_current_scope ())
  then true
  else
    if (Stack.length sym_stack) = 1
    then false
    else
      (Stack.push (Stack.pop sym_stack) temp_stack;
       lookup_sym sym temp_stack)

(* is the symbol defined in anything that is scope accessible? *)
let sym_exists sym =
  let temp_stack = Stack.create () in
  let result = lookup_sym sym temp_stack in
  let size = Stack.length temp_stack in
  let rec put_back_stacks len =
    if len = 0
    then result
    else
      (Stack.push (Stack.pop temp_stack) sym_stack;
       put_back_stacks (len - 1))
  in put_back_stacks size

(* add symbol to current scope *)
let add_sym sym =
  Stack.push (StringSet.add sym (Stack.pop sym_stack)) sym_stack

(* pull the string out of the OCaml data type *)
let get_symbol =
```

```
  function
  | Atom (Symbol sym) -> sym
  | _ -> raise (Failure "Can only get symbols from atoms")

(* keep track of any free variables we see in a lambda definition *)
let free_var_set = Stack.create ()

(* add a free variable to the stack *)
let add_sym_as_free_var sym = Stack.push sym free_var_set

(* get the length of a list (connected dotted pairs terminated with a nil) *)
let rec list_length =
  function
  | DottedPair (_, x) -> 1 + (list_length x)
  | Atom Nil -> 0
  | Atom _ -> raise (Failure "can't get length; not a list")

(* get the first element of a dotted pair *)
let car =
  function
  | DottedPair (x, _) -> x
  | _ ->
      raise
        (Failure
           "can not take the car of something that is not a dotted pair")

(* get the second element of a dotted pair *)
let cdr =
  function
```

```
  | DottedPair (_, x) -> x
  | _ ->
      raise
        (Failure
           "can not take the cdr of something that is not a dotted pair")

(*let is_cond sexpr =
    (car sexpr) = Atom(Symbol("cond"))*)
(* create C function declarator code snippet *)
let create_func_decl fun_name =
  "TAGGED_VALUE *" ^ (fun_name ^ "(TAGGED_VALUE*);\n")

(* a lambda has a parameter list, add those parameters to the current scope of this
lambda
    return a list with the strings representing those symbols for emission *)
let add_param_list_to_scope pl =
  let rec param_adder pl len param_list =
    if len > 0
    then
      (let sym = get_symbol (car pl)
       in (add_sym sym; param_adder (cdr pl) (len - 1) (sym :: param_list)))
    else param_list
  in param_adder pl (list_length pl) []

(* C function declarations string buffer *)
let func_decls = Buffer.create 1024

(* C function definitions string buffer *)
let func_defs = Buffer.create 1024
```

```
(* is this S-expression a lambda? *)
let is_lambda =
  function
  | DottedPair (s1, s2) ->
      (match s1 with | Atom (Symbol "lambda") -> true | _ -> false)
  | _ -> false

(* returns function (closure) that captures the prefix and increments the counter
*)
let sym_gen prefix =
  let count = ref (-1)
  in fun () -> (incr count; prefix ^ (string_of_int !count))

(*label generators *)
let eval_sym_gen = sym_gen "___eval_num_"

(* evaluation result *)
let lambda_sym_gen = sym_gen "___lambda_num_"

(* new function name *)
let cond_sym_gen = sym_gen "___cond_result_num_"

(* test condition result *)
let cond_end_sym_gen = sym_gen "COND_END_NUM_"

(* label to mark end of if statement *)
(* create a variable assignment snippet in C and return the variable generated so
we have
```

```
    a handle to it in things that reference it *)
let assign_str sym_generator buf =
  let new_sym = sym_generator ()
  in (Buffer.add_string buf ("TAGGED_VALUE *" ^ (new_sym ^ " = ")); new_sym)

(* remove double quotes from symbols, not needed internally *)
let escape_symbol sym =
  let len = String.length sym
  in
    if (sym.[0] = '"') && (sym.[len - 1] = '"')
    then String.sub sym 1 (len - 2)
    else sym

(* emit code to quote an S-expression, in other words, suppress evaluation, just
create
    the data structures *)
let rec quote_sexpr sexpr buf =
  match sexpr with
  | (* handle atoms: symbols, ints, doubles, and nil *) Atom a ->
      (match a with
       | Symbol s ->
           Buffer.add_string buf
             ("alloc_symbol(\"" ^ ((escape_symbol s) ^ "\")"))
       | Int i ->
           Buffer.add_string buf ("alloc_int(" ^ ((string_of_int i) ^ ")"))
       | Double d ->
           Buffer.add_string buf
             ("alloc_double(" ^ ((string_of_float d) ^ ")"))
       | Nil -> Buffer.add_string buf "nil")
```

```
  | (* create a cons structure and recursively quote its S-expressions.  this can
end up
        being an actual dotted pair or a list or a mix of any of the above *)
      DottedPair (sexpr1, sexpr2) ->
      (Buffer.add_string buf "construct(";
       quote_sexpr sexpr1 buf;
       Buffer.add_string buf ",";
       quote_sexpr sexpr2 buf;
       Buffer.add_string buf ")")

let rec
  (* code to recursively eval each argument to a function and emit code in left-to-
right
    order.  we can't rely on C's function calling mechanism for this, undefined
order *)
  explode_args args len var_list tail_is_lambda formal_params buf =
  if (list_length args) > 0
  then (* has args? *)
    (let this_arg = car args in (* get 1st arg *)
     let arg_name = eval_sexpr this_arg tail_is_lambda formal_params buf
     in
       (* eval it recursively *)
       if len = 1
       then arg_name :: var_list
       else
         (* last one, return list of the C variable names they were given *)
         (* no, keep evaling *)
         explode_args (cdr args) (len - 1) (arg_name :: var_list)
           tail_is_lambda formal_params buf)
```

```
  else []
and
  (* emit code to evaluate an S-expression, mostly all the work is done here *)
  (* sexpr, expression to generate, tail_is_lambda tells us if this lambda is being
detached
from its parent scope (i.e. returned as a value), formal_params any parameters
defined by
the lambda (i.e. not free variables, that's how we tell the difference buf, the
string
buffer we are emitting code to, returns C var name of evaluation results for others
to
reference *)
  eval_sexpr sexpr tail_is_lambda formal_params buf =
  match sexpr with
  | (* S-expression is just an atom, straight forward to evaluate.  everything
evals to
       itself, except symbols, which are looked up in the environment *)
      Atom a ->
      (match a with
       | Symbol s ->
           if sym_exists s
           then (* check symbol table *)
             (if tail_is_lambda && (not (List.mem s formal_params))
              then
                (* defining closure, but this var, while accessible is free and
must be closed over *)
                add_sym_as_free_var s
              else ();
              (let ns = assign_str eval_sym_gen buf
```

```
              in
                (Buffer.add_string buf
                   ("lookup_symbol(alloc_symbol(\"" ^
                      ((escape_symbol s) ^ "\"),this);\n"));
                 ns)))
          else raise (Failure ("symbol \"" ^ (s ^ "\" has not been bound")))
       | Int i ->
          let ns = assign_str eval_sym_gen buf
          in
            (Buffer.add_string buf
               ("alloc_int(" ^ ((string_of_int i) ^ ");\n"));
             ns)
       | Double d ->
          let ns = assign_str eval_sym_gen buf
          in
            (Buffer.add_string buf
               ("alloc_double(" ^ ((string_of_float d) ^ ");\n"));
             ns)
       | Nil ->
          let ns = assign_str eval_sym_gen buf
          in (Buffer.add_string buf "nil;\n"; ns))
  | (* To eval a list apply the function given by the car of the list.  if it's a
       built-in func call it, if it a special form handle it specially, if it's a
       lambda form then handle it really specially (will need to define a lambda's
code
       before calling it).  anything else is garbage. *)
      DottedPair (sexpr1, sexpr2) ->
      (match sexpr1 with
       | Atom a ->
```

```
            (match a with
             | (* three cases for the symbol: built-in function, special form, user
defined *)
                 Symbol s ->
                 if StringMap.mem s builtin_functions
                 then (* is built-in function? *)
                   (let num_params = list_length sexpr2 in
                    (* get num of params *)
                    let (Function (_, min_params, max_params)) =
                      StringMap.find s builtin_functions
                    in
                      if
                        (num_params >= min_params) &&
                          (num_params <= max_params)
                      then (* check arity of function against # params *)
                        (* evaluate the args left-to-right before calling function
*)
                        (let var_names =
                           explode_args sexpr2 num_params [] tail_is_lambda
                             formal_params buf in
                         let ns = assign_str eval_sym_gen buf
                         in
                           (* emit code to call built-in func *)
                           (Buffer.add_string buf
                              ("call_lambda(lookup_symbol(alloc_symbol(\"" ^
                                 (s ^ "\"),this),"));
                            Buffer.add_string buf (string_of_int num_params);
                            List.iter
                              (fun arg_str ->
```

```
                  Buffer.add_string buf ("," ^ arg_str))
              (List.rev var_names);
            Buffer.add_string buf ");\n";
            if tail_is_lambda
            then add_sym_as_free_var s
            else ();
            (* if in closure def, this symbol is a free var *)
            ns))
      else
        raise
          (Failure
             ("wrong number of parameters (" ^
                ((string_of_int num_params) ^
                   (") given to built-in function \"" ^
                      (s ^ "\""))))))
else
  (* is it a special form? (quote,define,lambda definition)*)
  if StringSet.mem s special_forms
  then
    (match s with
     | "quote" ->
         let num_params = list_length sexpr2
         in
           if num_params = 1
           then (* check # params *)
             (* okay, quote it and emit code, in function above *)
             (let ns = assign_str eval_sym_gen buf
              in
                (quote_sexpr (car sexpr2) buf;
```

```
                                Buffer.add_string buf ";\n";
                                ns))
                          else
                            raise
                              (Failure
                                 ("wrong number of parameters (" ^
                                    ((string_of_int num_params) ^
                                       ") given to quote special form")))
                    | "define" ->
                        let num_params = list_length sexpr2
                        in
                          if num_params = 2
                          then (* check # params *)
                            (* suppress evaluation of first param, it should be a
symbol *)
                            (let bind_sym =
                               try get_symbol (car sexpr2)
                               with
                               | _ ->
                                   raise
                                     (Failure
                                        "Can only bind values to symbols using
\"define\"")
                             in
                              (if sym_exists bind_sym
                               then ()
                               else add_sym bind_sym;
                               (* add the symbol to the symbol table *)
                               (* eval second param, if it's a lambda, create
```

```
closure *)
                                let evaled_expr =
                                  eval_sexpr (car (cdr sexpr2))
                                    ((is_lambda (car (cdr sexpr2))) ||
                                       tail_is_lambda)
                                    formal_params buf in
                                let ns = assign_str eval_sym_gen buf
                                in
                                  (Buffer.add_string buf
                                     ("add_binding(alloc_symbol(\"" ^
                                        (bind_sym ^
                                           ("\")," ^
                                              (evaled_expr ^ ",this);\n"))));
                                   ns)))
                            else
                              raise
                                (Failure
                                   ("wrong number of parameters (" ^
                                      ((string_of_int num_params) ^
                                         ") given to define special form")))
                      | "cond" ->
                          let cond_end_label = cond_end_sym_gen () in
                          (* get a new label to jump to the end of the if cases *)
                          let result_name = cond_sym_gen ()
                          in
                            (* variable to store the result of the evaluation *)
                            (Buffer.add_string buf
                               ("TAGGED_VALUE *" ^ (result_name ^ " = nil;\n"));
                             (* if no cases match, evals to false *)
```

```
                        let rec iter_cases cases len =
                          (* iterate through the cases *)
                          if len = 0
                          then
                            Buffer.add_string buf
                              (cond_end_label ^ ":\n")
                          else
                            (* add end label to the end of the case listings *)
                            (let this_case = car cases
                             in
                               if (list_length this_case) < 2
                               then (* check case param # *)
                                 raise
                                   (Failure
                                      "cond cases must have a test and at least
one consequent")
                               else
                                 (let test_case = car this_case in
                                  let test_result =
                                    eval_sexpr test_case tail_is_lambda
                                      formal_params buf
                                  in
                                    (* eval test condition *)
                                    (Buffer.add_string buf
                                       ("if(nil != " ^
                                          (test_result ^ ")\n{\n"));
                                     (* code to emit, if this case pases *)
                                     let consequents = cdr this_case in
                                     (* get consequents of test case *)
```

```
                                        let rec
                                          iter_conseq consequents len =
                                          (* for each consequent ... *)
                                          let consequent =
                                            car consequents in
                                          (* eval it, it the last one is a lambda,
create closure *)
                                          let conseq_val =
                                            eval_sexpr consequent
                                              (((len = 1) &&
                                                  (is_lambda consequent))
                                                 || tail_is_lambda)
                                              formal_params buf
                                          in
                                            if len = 1
                                            then
                                              (* emit code for consequents, this
case ran, jump to end label is emitted *)
                                              Buffer.add_string buf
                                                (result_name ^
                                                   (" = " ^
                                                      (conseq_val ^
                                                         (";\n" ^
                                                            ("goto " ^
                                                               (cond_end_label
                                                                  ^ ";\n}\n"))))))
                                            else
                                              iter_conseq (cdr consequents)
                                                (len - 1)
```

```
                                    in
                                      (iter_conseq consequents
                                         (list_length consequents);
                                       iter_cases (cdr cases) (len - 1)))))
                              in
                                (iter_cases sexpr2 (list_length sexpr2);
                                 result_name))
                        | "lambda" ->
                            (create_new_scope ();
                             (* lambda's create new lexical scope *)
                             let new_lambda_code_buf = Buffer.create 2048 in
                             (* generate the code for this function in a clean buffer
*)
                             let lambda_name = lambda_sym_gen () in
                             (* get a new lambda label name *)
                             let params = car sexpr2 in
                             (* get list of paramaters to lambda *)
                             let param_list = add_param_list_to_scope params in
                             (* add them to the scope and return an ocaml list of
params *)
                             let body = cdr sexpr2 in
                             let num_params = list_length params
                             in
                               (Buffer.add_string new_lambda_code_buf
                                  ("TAGGED_VALUE *" ^
                                     (lambda_name ^ "(TAGGED_VALUE *this)\n{\n"));
                                (* add label here for tail-call optimization *)
                                let rec iter_sexprs exprs len =
                                  (* for each expression in the body ... *)
```

```
                            (* if this is the last expresion and it's a lambda,
create closure *)
                            let tail_is_lambda =
                              ((len = 1) && (is_lambda (car exprs))) ||
                                tail_is_lambda in
                            let expr = car exprs in
                            let this_eval =
                              eval_sexpr expr tail_is_lambda param_list
                                new_lambda_code_buf
                            in
                              if len = 1
                              then
                                (* just eval'ed last expression in the lambda *)
                                (* return the last value, add to to the function
definition and declaration string buffers *)
                                (Buffer.add_string new_lambda_code_buf
                                   ("return " ^ (this_eval ^ ";\n}\n"));
                                 Buffer.add_string func_decls
                                   (create_func_decl lambda_name);
                                 Buffer.add_buffer func_defs
                                   new_lambda_code_buf)
                              else iter_sexprs (cdr exprs) (len - 1)
                          in
                            (iter_sexprs body (list_length body);
                             let ns = assign_str eval_sym_gen buf in
                             (* create list of parameters for emitting during
lambda type allocation *)
                             let alloc_param_snippet =
                               let rec alloc_param_gen params len str =
```

```
                    if len = 0
                    then str
                    else
                      (let this_param =
                         get_symbol (car params)
                       in
                         alloc_param_gen (cdr params)
                           (len - 1)
                           (str ^
                              (",\"" ^
                                 ((escape_symbol this_param)
                                    ^ "\""))))
                  in alloc_param_gen params num_params "" in
                let free_vars =
                  (* create set of free variables for emitting
during lambda type allocation *)
                  let rec free_vars_iter fv_set len =
                    if len <> 0
                    then
                      free_vars_iter
                        (StringSet.add
                           (Stack.pop free_var_set) fv_set)
                        (len - 1)
                    else fv_set
                  in
                    free_vars_iter StringSet.empty
                      (Stack.length free_var_set)
                in
                  (* emit code to generate lambda allocation *)
```

```
                                (Buffer.add_string buf
                                   ("alloc_lambda(" ^
                                      (lambda_name ^
                                         ("," ^
                                            ((string_of_int num_params) ^
                                               ("," ^
                                                  ("alloc_params(" ^
                                                     ((string_of_int
                                                         num_params)
                                                        ^
                                                        (alloc_param_snippet
                                                           ^
                                                           ("),(" ^
                                                              ((if
                                                                  (StringSet.
                                                                  cardinal
                                                                  free_vars)
                                                                  > 0
                                                                then "NULL"
                                                                else "this")
                                                                ^
                                                                ("," ^

("create_list_of_bindings("
                                                                  ^
                                                                  (string_of_int
                                                                  (StringSet.
                                                                  cardinal
```

```
free_vars)))))))))))));
                                  StringSet.iter
                                    (fun e ->
                                       Buffer.add_string buf
                                         (",lookup_binding(alloc_symbol(\""
                                            ^ (e ^ "\"),this)")))
                                    free_vars;
                                  Buffer.add_string buf "),false);\n";
                                  ignore (remove_scope ());
                                  ns))))
                      | _ ->
                          raise
                            (Failure
                               "should never get here.  just keeping ocaml happy."))
                  else (* must be a user defined func *)
                    if sym_exists s
                    then (* check that this symbol is in scope *)
                      (let num_params = list_length sexpr2 in
                       (* get the number of params, it's dynamic now, so no
checking *)
                       let var_names =
                         explode_args sexpr2 num_params [] tail_is_lambda
                           formal_params buf in
                       (* eval parameters first *)
                       let ns = assign_str eval_sym_gen buf
                       in
                         (* lookup symbol to get function, then call it, call
generation *)
                         (* number of params being passed in and a list of them *)
```

```
                        (Buffer.add_string buf
                           ("call_lambda(lookup_symbol(alloc_symbol(\"" ^
                              ((escape_symbol s) ^ "\"),this),"));
                         Buffer.add_string buf (string_of_int num_params);
                         List.iter
                           (fun arg_str ->
                              Buffer.add_string buf ("," ^ arg_str))
                           (List.rev var_names);
                         Buffer.add_string buf ");\n";
                         ns))
                    else
                      raise
                        (Failure
                           ("function call can't be made because \"" ^
                              (s ^ "\" is not defined")))
            | (* these others make no sense as the first element of a list being
evaluated *)
                Int i ->
                raise (Failure "integer is not function application")
            | Double d ->
                raise (Failure "double is not function application")
            | Nil -> raise (Failure "nil is not function application"))
       | (* is the first element of the list a list?  if so, only makes sense if
that list is a lambda (anonymous function def, followed by application) *)
           DottedPair (sexpr3, sexpr4) ->
           (match sexpr3 with
            | Atom (Symbol "lambda") ->
                let lambda_expr =
                  eval_sexpr sexpr1 tail_is_lambda formal_params buf in
```

```
                let num_params = list_length sexpr2 in
                let num_lambda_defined_params = list_length (car sexpr4)
                in
                  if num_params <> num_lambda_defined_params
                  then
                    raise
                      (Failure
                         ("wrong number of parameters (" ^
                            ((string_of_int num_params) ^
                               ") given to lambda function")))
                  else
                    (let var_names =
                       explode_args sexpr2 num_params [] tail_is_lambda
                         formal_params buf in
                     let ns = assign_str eval_sym_gen buf
                     in
                       (Buffer.add_string buf
                          ("call_lambda(" ^ (lambda_expr ^ ","));
                        Buffer.add_string buf (string_of_int num_params);
                        List.iter
                          (fun arg_str ->
                             Buffer.add_string buf ("," ^ arg_str))
                          (List.rev var_names);
                        Buffer.add_string buf ");\n";
                        ns))
            | _ ->
                (* first element is of the list is a list, which hopeful returns a
lambda when evaled, eval it, call it, and pray for the best *)
                let returns_lambda =
```

```
                  eval_sexpr sexpr1 tail_is_lambda formal_params buf in
                let num_params = list_length sexpr2 in
                (* get the number of params, it's dynamic now, so no checking *)
                let var_names =
                  explode_args sexpr2 num_params [] tail_is_lambda
                    formal_params buf in
                (* eval parameters first *)
                let ns = assign_str eval_sym_gen buf
                in
                  (* lookup symbol to get function, then call it, call generation
*)
                  (* number of params being passed in and a list of them *)
                  (Buffer.add_string buf
                     ("call_lambda(" ^ (returns_lambda ^ ","));
                   Buffer.add_string buf (string_of_int num_params);
                   List.iter
                     (fun arg_str -> Buffer.add_string buf ("," ^ arg_str))
                     (List.rev var_names);
                   Buffer.add_string buf ");\n";
                   ns)))

(* C file top header code *)
let main_header () =
  "#include <stdio.h>\n\
     #include <stdlib.h>\n\
     #include <stdbool.h>\n\
     #include \"init.h\"\n\
     #include \"memory.h\"\n\
     #include \"builtin.h\"\n\
```

```
     #include \"structs.h\"\n\
     #include \"functions.h\"\n\
     #include \"environment.h\"\n\n\n\
     extern ENV *global_env;\n\
     extern TAGGED_VALUE *t;\n\
     extern TAGGED_VALUE *nil;\n\n"
    ^
    ((Buffer.contents func_decls) ^
       ("\n\n" ^
          ((Buffer.contents func_defs) ^
             "\n\nint main(int argc, char* argv[])\n{\n\
     TAGGED_VALUE *this = NULL; /* no enclosing lambda, at top level */\n\n\
     init(); /* create primordial environment */\n\n")))

(* C file footer code *)
let main_footer () =
  "\ngc_on_exit(); /* clean up all objects, no marky... just sweepy */\n\
     clear_global_env(); /* clean up any bindings left in the global env with refs
to nowhere... */\n\
     \n\nreturn 0;\n}\n"

(* the main deal, walk the AST and generate C code evaluate each expression in the
list
    obtained by the parser (corresponding to toplevel lines in the source file).
    each expression can be evaluated recursively.
*)
let compile prog = (* create global scope *)
  (* add global constants and built-in functions to global scope *)
  (create_new_scope ();
```

```
add_sym "t";
add_sym "nil";
StringMap.iter (fun k v -> add_sym k) builtin_functions;
let (Prog all_exprs) = prog in
(* each line in the program (a s-expression) *)
let main_code = Buffer.create (16 * 1024)
in
  (* create main code buffer *)
  try
    ((* for each line... eval it, add garbage collection call *)
     List.iter
       (fun expr ->
          (ignore (eval_sexpr expr false [] main_code);
           ignore (Buffer.add_string main_code "gc();\n\n")))
       all_exprs;
     print_string
       ((main_header ()) ^
          ((Buffer.contents main_code) ^ (main_footer ()))))
  with
  | (* print main header, body, footer *) e ->
      (print_string
         ("FIND ERROR:\n" ^
            ((main_header ()) ^
               ((Buffer.contents main_code) ^ (main_footer ()))));
       raise e))
```

8.1.6 main.ml

The compiler's starting point and pretty printer.

```
open Ast

type pprint_opt = | Debug | Normal

type action = | PPrint of pprint_opt | Compile

(* pretty print atoms *)
let ppatom p a =
  match p with
  | Symbol s ->
      (match a with
       | Debug -> print_string ("sym:" ^ s)
       | Normal -> print_string s)
  | Int i ->
      (match a with
       | Debug -> (print_string "int:"; print_int i)
       | Normal -> print_int i)
  | Double d ->
      (match a with
       | Debug -> (print_string "dbl:"; print_float d)
       | Normal -> print_float d)
  | Nil ->
      (match a with
       | Debug -> print_string "sym:nil"
       | Normal -> print_string "nil")

(* pretty print S-expressions *)
let rec ppsexpr p islist act =
  match p with
```

```
  | Atom a -> ppatom a act
  | DottedPair (se1, se2) ->
      (if islist then () else print_string "( ";
       (match se1 with | Atom a -> ppatom a act | _ -> ppsexpr se1 false act);
       print_char ' ';
       (match se2 with
        | Atom Nil -> print_char ')'
        | Atom ((_ as a)) ->
            (print_string ". "; ppatom a act; print_string " )")
        | _ -> ppsexpr se2 true act))

(* pretty print the program *)
let pprint p a =
  match p with
  | Prog ss ->
      List.iter
        (fun i -> (ppsexpr i false a; print_newline (); print_newline ())) ss

(* starting point *)
let _ =
  let action =
    if (Array.length Sys.argv) > 1
    then
      (try
         List.assoc Sys.argv.(1)
           [ ("-d", (PPrint Debug)); ("-p", (PPrint Normal)) ]
       with | Not_found -> Compile)
    else Compile in
  let lexbuf = Lexing.from_channel stdin in
```

```
let prog = Parser.prog Scanner.token lexbuf
in
  match action with
  | Compile -> Compiler.compile prog
  | PPrint a -> pprint prog a
```

8.2 C Runtime Library

8.2.1 builtin.h

Built-in functions header.

```
#ifndef __BUILTIN_H__
#define __BUILTIN_H__

TAGGED_VALUE *atomp(TAGGED_VALUE *);
TAGGED_VALUE *symbolp(TAGGED_VALUE *);
TAGGED_VALUE *eqp(TAGGED_VALUE *);
TAGGED_VALUE *equal(TAGGED_VALUE *);
TAGGED_VALUE *lessthan(TAGGED_VALUE *);
TAGGED_VALUE *greaterthan(TAGGED_VALUE *);
TAGGED_VALUE *lessthanequal(TAGGED_VALUE *);
TAGGED_VALUE *greaterthanequal(TAGGED_VALUE *);
TAGGED_VALUE *add(TAGGED_VALUE *);
TAGGED_VALUE *sub(TAGGED_VALUE *);
TAGGED_VALUE *prod(TAGGED_VALUE *);
TAGGED_VALUE *divide(TAGGED_VALUE *);
TAGGED_VALUE *mod(TAGGED_VALUE *);
TAGGED_VALUE *expt(TAGGED_VALUE *);
TAGGED_VALUE *log_num_base(TAGGED_VALUE *);
TAGGED_VALUE *cons(TAGGED_VALUE *);
TAGGED_VALUE *first(TAGGED_VALUE *);
TAGGED_VALUE *rest(TAGGED_VALUE *);
TAGGED_VALUE *read_inp(TAGGED_VALUE *);
TAGGED_VALUE *print(TAGGED_VALUE *);
TAGGED_VALUE *implode(TAGGED_VALUE *);
TAGGED_VALUE *explode(TAGGED_VALUE *);

#endif /* __BUILTIN_H__ */
```

8.2.2 builtin.c

Built-in functions source.

```c
#include <stdio.h>
#include <math.h>
#include <ctype.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "init.h"
#include "error.h"
#include "structs.h"
#include "functions.h"
#include "environment.h"

typedef enum e_token { END, UNKNOWN, LPAR, RPAR, SYM, INUM, FNUM, DOT } TOKEN;

static TAGGED_VALUE *create_read_tv();
static TAGGED_VALUE *create_read_list_tv();

static char user_input_buffer[4096];
static int uibi = 0;   /* index into user input buffer (what the user entered) */

extern TAGGED_VALUE *t;
extern TAGGED_VALUE *nil;

/* atom? impl */
TAGGED_VALUE *atomp(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);

    if(!IS_CONS(arg1))
    {
        return t;
    }
    else
    {
        return nil;
    }
}

/* symbol? impl */
TAGGED_VALUE *symbolp(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);

    if(IS_SYMBOL(arg1))
    {
        return t;
    }
    else
    {
        return nil;
    }
```

```
}

/* eql? impl. */
TAGGED_VALUE *eqp(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);
    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(arg1->type != arg2->type)  /* types must be the same */
    {
        return nil;
    }
    else if(IS_NUMBER(arg1))
    {
        if(GET_NUM_VAL(arg1) == GET_NUM_VAL(arg2)) /* are numbers equal */
            return t;
        else
            return nil;
    }
    else if(IS_SYMBOL(arg1))
    {
        if(0 == strcmp(arg1->string_val, arg2->string_val)) /* are symbols eviq.
            */
            return t;
        else
            return nil;
    }
    else if(IS_LAMBDA(arg1))
    {
        if(arg1->lambda.func_name == arg2->lambda.func_name) /* do they point to
            the same func code */
            return t;
        else
            return nil;
    }
    else /* IS_CONS */     /* recursively check these conditions by
        deconstructing the list item by item */
    {
        add_binding(alloc_symbol(ARG_PARAM_LIST),
                    construct(car(arg1), construct(car(arg2), nil)), this);
        add_binding(alloc_symbol(ARG_PARAM_COUNT), alloc_int(2), this);
        TAGGED_VALUE *result_car = eqp(this);

        add_binding(alloc_symbol(ARG_PARAM_LIST),
                    construct(cdr(arg1), construct(cdr(arg2), nil)), this);
        add_binding(alloc_symbol(ARG_PARAM_COUNT), alloc_int(2), this);
        TAGGED_VALUE *result_cdr = eqp(this);

        if(result_car == result_cdr)
            return t;
        else
            return nil;
    }

}
```

```
/* equal? impl */
TAGGED_VALUE *equal(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail("= can only operate on numeric types.");
    }

    if(GET_NUM_VAL(arg1) == GET_NUM_VAL(arg2))
        return t;
    else
        return nil;
}

/* < impl */
TAGGED_VALUE *lessthan(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail("< can only operate on numeric types.");
    }

    if(GET_NUM_VAL(arg1) < GET_NUM_VAL(arg2))
        return t;
    else
        return nil;
}

/* > impl */
TAGGED_VALUE *greaterthan(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail("> can only operate on numeric types.");
    }

    if(GET_NUM_VAL(arg1) > GET_NUM_VAL(arg2))
        return t;
    else
        return nil;
```

```
}

/* <= impl */
TAGGED_VALUE *lessthanequal(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail("<= can only operate on numeric types.");
    }

    if(GET_NUM_VAL(arg1) <= arg2->int_val)
        return t;
    else
        return nil;
}

/* >= impl */
TAGGED_VALUE *greaterthanequal(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail(">= can only operate on numeric types.");
    }

    if(GET_NUM_VAL(arg1) >= GET_NUM_VAL(arg2))
        return t;
    else
        return nil;
}

/* + impl */
TAGGED_VALUE *add(TAGGED_VALUE *this)
{
    int int_sum = 0;
    double double_sum = 0.0;
    bool saw_double = false;

    /* takes multiple var args */
    TAGGED_VALUE *arg_count = lookup_symbol(alloc_symbol(ARG_PARAM_COUNT), this);
    int num_args = arg_count->int_val;
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    /* keep sum, keeping track of whether the result can be an integer or must be
        double */
    for(int i=0; i<num_args; i++)
    {
```

```
            TAGGED_VALUE *val = car(arg_list);
            arg_list = cdr(arg_list);
            switch(val->type)
            {
                case INT:
                    int_sum += val->int_val;
                    break;
                case FLOAT:
                    double_sum += val->double_val;
                    saw_double = true;
                    break;
                default:
                    fail("Can not add something that is not a number.");
                    break;
            }
        }

        TAGGED_VALUE *final_sum;
        if(saw_double)
        {
            final_sum = alloc_double((double)int_sum + double_sum);
        }
        else
        {
            final_sum = alloc_int(int_sum);
        }

        return final_sum;
    }

    /* - impl */
    TAGGED_VALUE *sub(TAGGED_VALUE *this)
    {
        int int_result = 0;
        double double_result = 0.0;
        bool saw_double = false;

        /* takes multiple var args */
        TAGGED_VALUE *arg_count = lookup_symbol(alloc_symbol(ARG_PARAM_COUNT), this);
        int num_args = arg_count->int_val;
        TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

        /* keep total, keeping track of whether the result can be an integer or must
            be double */
        for(int i = 0; i < num_args; i++)
        {
            TAGGED_VALUE *val = car(arg_list);
            arg_list = cdr(arg_list);

            if(saw_double)
            {
                if(IS_NUMBER(val))
                {
                    double_result -= GET_NUM_VAL(val);
                }
                else
```

```
            {
                fail("Sub can only work with numerical values.");
            }
        }
        else
        {
            if(IS_INT(val))
            {
                if(0 == i && num_args > 1)
                {
                    int_result = GET_NUM_VAL(val);
                }
                else
                {
                    int_result -= GET_NUM_VAL(val);
                }
            }
            else if(IS_FLOAT(val))
            {
                saw_double = true;

                if(0 == i && num_args > 1)
                {
                    double_result = GET_NUM_VAL(val);
                }
                else
                {
                    double_result = (double)(int_result) - GET_NUM_VAL(val);
                }
            }
            else
            {
                fail("Sub can only work with numerical values.");
            }
        }
    }

    if(saw_double)
        return alloc_double(double_result);
    else
        return alloc_int(int_result);
}

/* prod impl */
TAGGED_VALUE *prod(TAGGED_VALUE *this)
{
    int int_prod = 1;
    double double_prod = 1.0;
    bool saw_double = false;

    /* takes multiple var args */
    TAGGED_VALUE *arg_count = lookup_symbol(alloc_symbol(ARG_PARAM_COUNT), this);
    int num_args = arg_count->int_val;
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    for(int i=0; i<num_args; i++)
```

```
    {
        TAGGED_VALUE *val;
        val = car(arg_list);
        arg_list = cdr(arg_list);
        switch(val->type)
        {
            case INT:
                int_prod *= val->int_val;
                break;
            case FLOAT:
                double_prod *= val->double_val;
                saw_double = true;
                break;
            default:
                fail("Cannot multiply something that is not a number.");
                break;
        }
    }

    TAGGED_VALUE *final_prod;
    if(saw_double)
    {
        final_prod = alloc_double((double)int_prod + double_prod);
    }
    else
    {
        final_prod = alloc_int(int_prod);
    }

    return final_prod;
}

/* / impl */
TAGGED_VALUE *divide(TAGGED_VALUE *this)
{
    int int_result = 0;
    double double_result = 0.0;
    bool is_double_result = false;

    /* takes multiple var args */
    TAGGED_VALUE *arg_count = lookup_symbol(alloc_symbol(ARG_PARAM_COUNT), this);
    int num_args = arg_count->int_val;
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    if(1 == num_args)
    {
        TAGGED_VALUE *val = car(arg_list);
        if(IS_INT(val) && 1 == val->int_val) /* one is only divisble by itself */
        {
            return val;
        }
        else if(IS_NUMBER(val)) /* anything else will be a double */
        {
            return alloc_double(1.0/GET_NUM_VAL(val));
        }
        else
```

```
            fail("\"/\" can only work with numerical values.");
        }
        else
        {
            for(int i = 0; i < num_args; i++)
            {
                TAGGED_VALUE *val = car(arg_list);
                arg_list = cdr(arg_list);

                if(is_double_result)
                {
                    if(IS_NUMBER(val))
                    {
                        double_result /= GET_NUM_VAL(val);
                    }
                    else
                    {
                        fail("\"/\" can only work with numerical values.");
                    }
                }
                else
                {
                    if(IS_INT(val) && 0 == (int_result % val->int_val))
                    {
                        if(0 == i)
                        {
                            int_result = GET_NUM_VAL(val);
                        }
                        else
                        {
                            int_result /= GET_NUM_VAL(val);
                        }
                    }
                    else if(IS_NUMBER(val))
                    {
                        is_double_result = true;

                        if(0 == i)
                        {
                            double_result = GET_NUM_VAL(val);
                        }
                        else
                        {
                            double_result = (double)(int_result) / GET_NUM_VAL(val);
                        }
                    }
                    else
                    {
                        fail("\"/\" can only work with numerical values.");
                    }
                }
            }
        }

        if(is_double_result)
            return alloc_double(double_result);
```

```
    else
        return alloc_int(int_result);
}

/* mod impl */
TAGGED_VALUE *mod(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail("mod can only operate on numeric types.");
    }

    if(IS_INT(arg1) && IS_INT(arg2)) /* int mod */
    {
        return alloc_int(arg1->int_val % arg2->int_val);
    }
    else /* float mod */
    {
        return alloc_double(fmod(GET_NUM_VAL(arg1),GET_NUM_VAL(arg2)));
    }
}

/* expt impl */
TAGGED_VALUE *expt(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail("expt can only operate on numeric types.");
    }

    return alloc_double(pow(GET_NUM_VAL(arg1), GET_NUM_VAL(arg2)));
}

/* log impl */
TAGGED_VALUE *log_num_base(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    if(!IS_NUMBER(arg1) || !IS_NUMBER(arg2))
    {
        fail("log can only operate on numeric types.");
    }
```

```
    /* (log n / log base) */
    return alloc_double(log(GET_NUM_VAL(arg1)) / log(GET_NUM_VAL(arg2)));
}

/* cons impl */
TAGGED_VALUE *cons(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);
    TAGGED_VALUE *arg2 = car(cdr(arg_list));

    return construct(arg1, arg2);
}

/* first impl */
TAGGED_VALUE *first(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);

    if(!IS_CONS(arg1) && !(IS_SYMBOL(arg1) && nil == arg1))
    {
        fail("first can only work on a cons cell.");
    }

    if(IS_SYMBOL(arg1))
        return nil;
    else
        return arg1->cell.car;
}

/* rest impl */
TAGGED_VALUE *rest(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);

    if(!IS_CONS(arg1) && !(IS_SYMBOL(arg1) && nil == arg1))
    {
        fail("rest can only work on a cons cell.");
    }

    if(IS_SYMBOL(arg1))
        return nil;
    else
        return arg1->cell.cdr;
}

/* implode impl */
TAGGED_VALUE *implode(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);
    /* argument is a list, but it was wrapped in a list by call_lambda,
```

```
        so take it out of that list first */
    arg_list = car(arg_list);
    int list_len = list_length(arg_list);

    char *str_buffer = NULL;
    do
    {
        TAGGED_VALUE *current_arg = car(arg_list);

        if(!IS_SYMBOL(current_arg))
        {
            fail("Cannot impolde on anything but symbols.");
        }

        str_buffer = realloc(str_buffer,
                        (sizeof(char *)*strlen(current_arg->string_val)
                         + ((str_buffer)?strlen(str_buffer) : 0)+1));
        if(NULL == str_buffer)
        {
            fail("Could not allocate buffer storage during implode function.");
        }

        strcat(str_buffer, current_arg->string_val);


        arg_list = cdr(arg_list);
        list_len--;
    } while(list_len > 0);

    TAGGED_VALUE *result = alloc_symbol(str_buffer);
    free(str_buffer);

    return result;
}

/* explode impl */
TAGGED_VALUE *explode(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);

    int length = (int)strlen(arg1->string_val);

    char temp_buf[2] = {'\0','\0'};

    temp_buf[0] = arg1->string_val[length-1];

    /* alloc symbol for each character in symbol and add to a list to return */
    TAGGED_VALUE *result =
        construct(alloc_symbol(temp_buf), nil);
    for(int i = length-2; i >= 0; i--)
    {
        temp_buf[0] = arg1->string_val[i];
        result = construct(alloc_symbol(temp_buf), result);
    }
```

```
    return result;
}

static bool is_token_sep(char c)
{
    return (isspace(c) || '(' == c || ')' == c);
}

static bool is_symbol_char(char c)
{
    char sym_chars[28] = {
        '~','!','@','#','$','%','^','&','*','+','=',':','?','/','<','>',
        '.',',','-','+','{','}','[',']','|','\\','\'','`'
    };

    if(isalpha(c))
        return true;

    for(int i = 0; i < 28; i++)
    {
        if(sym_chars[i] == c)
            return true;
    }

    return false;
}

static TOKEN get_next_token(char *buff)
{
    int i = 0;      /* token buffer index */

    char in_c = user_input_buffer[uibi++];  /* get the first char */
    if('\n' == in_c) return END;

    while(isspace(in_c))        /* read through white space */
    {
        in_c = user_input_buffer[uibi++];
        if('\n' == in_c) return END;
    }

    if('.' == in_c)
    {
        in_c = user_input_buffer[uibi++];
        if(!is_token_sep(in_c))
            uibi--; /* unget */
        else
            return DOT;
    }
    if('-' == in_c || '+' == in_c || isdigit(in_c)) /* should be a number */
    {
        if('-' == in_c || '+' == in_c)
        {
            buff[i++] = in_c;
            in_c =user_input_buffer[uibi++];
            if(!isdigit(in_c))
```

```
            {
                uibi--; /* unget */
                uibi--;
                i--;
                goto symbol_check;
            }
        }
        do
        {
            buff[i++] = in_c;
            in_c = user_input_buffer[uibi++];       /* read it all in */
        } while(!is_token_sep(in_c) && '\n' != in_c);
        uibi--;  /* unget */

        buff[i] = '\0';

        char *endptr = NULL;
        strtol(buff, &endptr, 10);  /* if not, try an int (well, long really) */
        if(endptr == &buff[i])
            return INUM;

        endptr = NULL;
        strtod(buff, &endptr);      /* can it be fully converted to a double? */
        if(endptr == &buff[i])
            return FNUM;

        /* otherwise... error */    /* don't know what it is */
        return UNKNOWN;
    }
symbol_check:
    if(is_symbol_char(in_c))    /* token starts with a SYMBOL char */
    {
        do
        {
            buff[i++] = in_c;       /* read it all into the token buffer */
            in_c = user_input_buffer[uibi++];
        } while(!is_token_sep(in_c) && EOF != '\n');
        uibi--; /* unget */

        buff[i] = '\0';
        return SYM;     /* we found a symbol token */
    }
    if('\"' == in_c)    /* starts with a "... still a token, eat the "" */
    {
        in_c = user_input_buffer[uibi++]; /* skip the " */
        do
        {
            buff[i++] = in_c;
            in_c = user_input_buffer[uibi++];
        } while('\"' != in_c && '\n' != in_c);
        if('\n' == in_c) return END;
```



```
        buff[i] = '\0';
        return SYM;     /* a symbol token */
    }
    if('(' == in_c)
```

```
    {
        return LPAR;
    }
    if(')' == in_c)
    {
        return RPAR;
    }

    return UNKNOWN;
}

static TAGGED_VALUE *create_read_list_tv()
{
    char buff[4096];

    switch(get_next_token(buff))
    {
        case SYM:
            return construct(alloc_symbol(buff),create_read_list_tv());
            break;
        case INUM:
            return construct(alloc_int((int)strtol(buff, NULL, 10)),
                create_read_list_tv());
            break;
        case FNUM:
            return construct(alloc_double(strtod(buff, NULL)),create_read_list_tv
                ());
            break;
        case RPAR:
            return nil;
            break;
        case LPAR:
            return construct(create_read_list_tv(),create_read_list_tv());
            break;
        case DOT:
            return create_read_tv();
            break;
        case END:
        case UNKNOWN:
        default:
            fail("Bad user input given to read.\n");
            break;
    }

    fail("Bad user input given to read.\n");
    return NULL;
}

static TAGGED_VALUE *create_read_tv()
{
    char buff[4096];

    switch(get_next_token(buff))
    {
        case SYM:
            return alloc_symbol(buff);
```

```
            break;
        case INUM:
            return alloc_int((int)strtol(buff, NULL, 10));
            break;
        case FNUM:
            return alloc_double(strtod(buff, NULL));
            break;
        case LPAR:;
            return create_read_list_tv();
            break;
        case RPAR:
        case DOT:
        case END:
        case UNKNOWN:
        default:
            fail("Bad user input given to read.\n");
            break;
    }

    fail("Bad user input given to read.\n");
    return NULL;
}

/* read impl ... totally gross badly implemented scanner/parser, ick */
TAGGED_VALUE *read_inp(TAGGED_VALUE *this)
{
    uibi = 0;  /* reset index for new input */
    fgets(user_input_buffer, 4096, stdin);
    return create_read_tv();
}

/* print helper function to recursively print results */
static TAGGED_VALUE *print_imp(TAGGED_VALUE *tv, bool is_in_list, int depth)
{
    switch(tv->type)
    {
        case SYMBOL:
            printf("\"%s\"", tv->string_val);
            break;
        case INT:
            printf("%d", tv->int_val);
            break;
        case FLOAT:
            printf("%f", tv->double_val);
            break;
        case LAMBDA:
            printf("<lambda: %p>", tv->lambda.func_name);
            break;
        case CONS:
            if(!is_in_list)
            {
                printf("( ");
            }
            print_imp(tv->cell.car, false, depth+1);
            if(!IS_CONS(tv->cell.cdr))
            {
```

```
                if(nil != tv->cell.cdr)
                {
                    printf(" . ");
                    print_imp(tv->cell.cdr, false, depth+1);
                }
                printf(" )");
            }
            else
            {
                printf(" ");
                print_imp(tv->cell.cdr, true, depth+1);
            }
    }

    if(0 == depth)
    {
        printf("\n");
    }

    return tv;
}

/* print impl*/
TAGGED_VALUE *print(TAGGED_VALUE *this)
{
    TAGGED_VALUE *arg_list = lookup_symbol(alloc_symbol(ARG_PARAM_LIST), this);

    TAGGED_VALUE *arg1 = car(arg_list);

    return print_imp(arg1, false, 0);
}
```

8.2.3 environment.h

Environment data structures.

```
#ifndef __ENVIRONMENT_H__
#define __ENVIRONMENT_H__

#include "structs.h"

struct s_tagged_value;
typedef struct s_tagged_value TAGGED_VALUE;

typedef struct s_binding {
    struct s_tagged_value *symbol;
    struct s_tagged_value *value;
    struct s_binding *next;
} BINDING;

typedef struct s_env {
    BINDING *head;
    struct s_env *parent;
} ENV;

BINDING *alloc_binding(TAGGED_VALUE *, TAGGED_VALUE *);
ENV *alloc_environment(void);
TAGGED_VALUE *add_binding(TAGGED_VALUE *, TAGGED_VALUE *, TAGGED_VALUE *);
TAGGED_VALUE *lookup_symbol(TAGGED_VALUE *,TAGGED_VALUE *);
BINDING *lookup_binding(TAGGED_VALUE *, TAGGED_VALUE *);
void delete_env(void);
void delete_bindings(BINDING *);
void clear_global_env(void);

#endif /* __ENVIRONMENT_H__ */
```

8.2.4 environment.c

Environment source, functions and utilities.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "error.h"
#include "structs.h"
#include "environment.h"

static ENV g_env = {NULL, NULL};
const ENV *global_env = &g_env;
ENV *current_env = &g_env;


BINDING *alloc_binding(TAGGED_VALUE *symbol, TAGGED_VALUE *value)
{
    BINDING *new_binding = (BINDING *)malloc(sizeof(BINDING));
    if(NULL == new_binding)
    {
        fail("Problem allocating binding memory.");
    }

    new_binding->symbol = symbol;
    new_binding->value = value;
    new_binding->next = NULL;

    return new_binding;
}

ENV *alloc_environment()
{
    ENV *new_env = (ENV *)malloc(sizeof(ENV));
    if(NULL == new_env)
    {
        fail("Problem allocating environment memory.");
    }

    new_env->parent = current_env;
    new_env->head = NULL;

    current_env = new_env;

    return new_env;
}

static TAGGED_VALUE *add_binding_in_env(TAGGED_VALUE *symbol, TAGGED_VALUE *value
    , ENV *env)
{
    BINDING *current_bind = env->head;
    while(current_bind)     /* iterate through bindings */
    {
        /* is symbol the same? */
        if(0 == strcmp(current_bind->symbol->string_val, symbol->string_val))
        {
            current_bind->value = value;     /* update the value and stop */
            return value;
        }
```

```
        current_bind = current_bind->next;    /* next binding */
    }

    return NULL;
}

static TAGGED_VALUE *add_binding_in_closure(TAGGED_VALUE *symbol, TAGGED_VALUE *
    value,
                                            BINDING *closure_bindings)
{
    BINDING *current_bind = closure_bindings;
    while(current_bind)
    {
        /* is symbol the same? */
        if(0 == strcmp(current_bind->symbol->string_val, symbol->string_val))
        {
            current_bind->value = value;
            return value;
        }

        current_bind = current_bind->next;
    }

    return NULL;
}

TAGGED_VALUE *add_binding(TAGGED_VALUE *symbol, TAGGED_VALUE *value, TAGGED_VALUE
    *caller)
{
    if(SYMBOL != symbol->type)
    {
        fail("Not binding value to a symbol.");
    }

    if(NULL == caller)  /* special case, the global env is the scope */
    {
        TAGGED_VALUE * result =
            add_binding_in_env(symbol, value, global_env);

        if(NULL != result)
        {
            return result;
        }
    }
    else
    {
        TAGGED_VALUE *result;

        /* first check if the function has any variables in clousure */
        result =
         add_binding_in_closure(symbol, value, caller->lambda.closure_bindings);
```



```
        if(NULL != result)
        {
            return result;
        }
```

```
        /* check my static scope first for the binding */
        TAGGED_VALUE *scope_ptr = caller;
        while(NULL != scope_ptr)
        {
            result = add_binding_in_env(symbol, value, scope_ptr->lambda.
                running_env);

            if(NULL == result)
            {
                /* try next scope */
                scope_ptr = scope_ptr->lambda.parent_scope;
            }
            else
            {
                return result;
            }
        }
    }

    /* binding wasn't found in any existing environment; add it to the current
        one */
    /* pick either the current lambda's running environment or the global one */
    ENV *env_to_add_binding_to;
    if(NULL == caller)
    {
        env_to_add_binding_to = global_env;
    }
    else
    {
        env_to_add_binding_to = caller->lambda.running_env;
    }

    if(NULL == env_to_add_binding_to->head)   /* No symbols in the table yet */
    {
        /* alloc and insert binding */
        env_to_add_binding_to->head = alloc_binding(symbol, value);
    }
    else    /* add it as a new binding to existing caller's enviornment */
    {
        BINDING *next_bind = env_to_add_binding_to->head;
        /* add new binding */
        env_to_add_binding_to->head = alloc_binding(symbol, value);
        /* set next pointer to what head was pointing to */
        env_to_add_binding_to->head->next = next_bind;
    }

    return value;
}

static TAGGED_VALUE *lookup_symbol_in_env(TAGGED_VALUE *symbol, ENV *env)
{
    BINDING *current_bind = env->head;
    while(current_bind)     /* iterate through bindings */
    {
        /* is symbol the same? */
```

```
        if(0 == strcmp(current_bind->symbol->string_val, symbol->string_val))
        {
            return current_bind->value;
        }
        current_bind = current_bind->next;    /* next binding */
    }

    return NULL;
}

static TAGGED_VALUE *lookup_symbol_in_closure(TAGGED_VALUE *symbol,
                                              BINDING *closure_bindings)
{
    BINDING *current_bind = closure_bindings;

    while(current_bind)
    {
        /* is symbol the same? */
        if(0 == strcmp(current_bind->symbol->string_val, symbol->string_val))
        {
            return current_bind->value;
        }

        current_bind = current_bind->next;
    }

    return NULL;
}

TAGGED_VALUE *lookup_symbol(TAGGED_VALUE *symbol, TAGGED_VALUE *caller)
{
    if(SYMBOL != symbol->type)
    {
        fail("Can only lookup symbols.");
    }


    if(NULL == caller)  /* special case, this lookup is in the global env */
    {
        TAGGED_VALUE * result =
            lookup_symbol_in_env(symbol, global_env);

        if(NULL != result)
        {
            return result;
        }
    }
    else
    {
        TAGGED_VALUE *result = NULL;

        /* first check if the function has any variables in clousure */
        result =
        lookup_symbol_in_closure(symbol,caller->lambda.closure_bindings);

        if(NULL != result)
```

```
        {
            return result;
        }

        TAGGED_VALUE *scope_ptr = caller;

        /* check my calling environment first for symbol then go to
            my parent scope's environment and recursively search until
            we get to the global environment from there */
        while(NULL != scope_ptr)
        {
            result =
                lookup_symbol_in_env(symbol, scope_ptr->lambda.running_env);
            if(NULL == result)
            {
                /* try the next environment */
                scope_ptr = scope_ptr->lambda.parent_scope;
            }
            else
            {
                return result;
            }
        }

        /* no further parent calling enviroments, now check the global env */
        result = lookup_symbol_in_env(symbol, global_env);
        if(result != NULL)
        {
            return  result;
        }
    }

    /* didn't find it in any accessible scope's environment */
    char error_msg[1024];
    sprintf(error_msg, "Symbol \"%s\" is unbound.", symbol->string_val);
    fail(error_msg);
    return NULL;  /* won't get to here, shuts up compiler */
}

static BINDING *lookup_binding_in_env(TAGGED_VALUE *symbol, ENV *env)
{
    BINDING *current_bind = env->head;
    while(current_bind)     /* iterate through bindings */
    {
        /* is symbol the same? */
        if(0 == strcmp(current_bind->symbol->string_val, symbol->string_val))
        {
            return current_bind;
        }
        current_bind = current_bind->next;    /* next binding */
    }

    return NULL;
}

BINDING *lookup_binding(TAGGED_VALUE *symbol, TAGGED_VALUE *caller)
```

```
{
    if(SYMBOL != symbol->type)
    {
        fail("Can only lookup symbols.");
    }

    if(NULL == caller)  /* special case, this lookup is in the global env */
    {
        BINDING *result =
        lookup_binding_in_env(symbol, global_env);

        if(NULL != result)
        {
            return result;
        }
    }
    else
    {
        TAGGED_VALUE *scope_ptr = caller;

        /* check my calling environment first for symbol then go to
         my parent scope's environment and recursively search until
         we get to the global environment from there */
        while(NULL != scope_ptr)
        {
            BINDING *result =
            lookup_binding_in_env(symbol, scope_ptr->lambda.running_env);
            if(NULL == result)
            {
                /* try the next environment */
                scope_ptr = scope_ptr->lambda.parent_scope;
            }
            else
            {
                return result;
            }

        }
    }

    /* didn't find it in any accessible scope's environment */
    char error_msg[1024];
    sprintf(error_msg, "Symbol \"%s\" is unbound.", symbol->string_val);
    fail(error_msg);
    return NULL;  /* won't get to here, shuts up compiler */
}

void delete_bindings(BINDING *binding_head)
{
    BINDING *current_binding = binding_head;    /* start at head and traverse
        list */
    while(current_binding)
    {
        BINDING *binding_to_delete = current_binding;
        current_binding = current_binding->next;    /* advance current pointer */
        free(binding_to_delete);                    /* delete binding memory */
```

```
    }
}

void delete_env()
{
    if(current_env == global_env)
    {
        fail("Cannot destroy global environment.");
    }

    ENV *del_env = current_env;

    current_env = del_env->parent;

    delete_bindings(del_env->head);
    free(del_env);
}

void clear_global_env()
{
    delete_bindings(global_env->head);
}
```

8.2.5 error.h

Common error routine header.

```
#ifndef __ERROR_H__
#define __ERROR_H__

void fail(char *);

#endif  /* __ERROR_H__ */
```

8.2.6 error.c

Common error routine source.

```c
#include <stdio.h>
#include <stdlib.h>
#include "memory.h"


void fail(char *error_message)
{
    fprintf(stderr, "PLT: %s\n", error_message);
    gc_on_exit();
    exit(EXIT_FAILURE);
}
```

8.2.7 functions.h

Call lambda routine header.

```
#ifndef __FUNCTIONS_H__
#define __FUNCTIONS_H__

#define ARG_PARAM_LIST "param_list"
#define ARG_PARAM_COUNT "param_count"

TAGGED_VALUE *call_lambda(TAGGED_VALUE *, int, ...);

#endif /* __FUNCTIONS_H__ */
```

8.2.8 functions.c

Function to handle calling lambdas.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdarg.h>
#include "error.h"
#include "structs.h"
#include "functions.h"
#include "environment.h"

extern TAGGED_VALUE *nil;

TAGGED_VALUE *call_lambda(TAGGED_VALUE *tv, int num_params, ...)
{
    if(LAMBDA != tv->type)
    {
        fail("Illegal function call.");
    }

    if((!tv->lambda.is_builtin && num_params != tv->lambda.param_list.length)
        || (tv->lambda.is_builtin && num_params < tv->lambda.param_list.length))
    {
        char error_msg[1024];
        sprintf(error_msg, "Illegal function call, expecting %d params \
            called with %d params.\n", tv->lambda.param_list.length, num_params);
        fail(error_msg);
    }

    /* create new environment on the stack */
    ENV *lambda_env = alloc_environment();

    ENV *old_running_env = tv->lambda.running_env;  /* if recursive call, save
        current running env */
    tv->lambda.running_env = lambda_env;  /* set ref to it */

    /* get parameters, add all parameters to a special list and just add that
        list
     as a parameter with a special name if it a built-in function, otherwise,
     for regular function just add a binding to environment for each param */
    bool builtin_func = tv->lambda.is_builtin;

    TAGGED_VALUE *params_in_list = NULL;

    va_list params;

    va_start(params, num_params);
    for(int i = 0; i < num_params ; i++)
    {
        if(builtin_func)
        {
            if(i == 0)
            {
                params_in_list = construct(va_arg(params, TAGGED_VALUE *), nil);
            }
            else
            {
                append(va_arg(params, TAGGED_VALUE *), params_in_list);
```

```
            }
        }
        else
        {
            add_binding(tv->lambda.param_list.params[i],
                va_arg(params, TAGGED_VALUE *), tv);
        }
    }
    va_end(params);

    if(builtin_func)
    {
        add_binding(alloc_symbol(ARG_PARAM_LIST), params_in_list, tv);
        add_binding(alloc_symbol(ARG_PARAM_COUNT), alloc_int(num_params), tv);
    }

    /* execute lambda */
    TAGGED_VALUE *result = (*(tv->lambda.func_name))(tv);

    tv->lambda.running_env = old_running_env;  /* restore old running env */
    delete_env();  /* pop environment off the stack */

    return result;
}
```

8.2.9 init.h

Init file header.

```
#ifndef __INIT_H__
#define __INIT_H__

void init(void);

#endif  /* __INIT_H__ */
```

8.2.10 init.c

Initializes global environment before compiled code runs.

```c
#include <stdbool.h>
#include <stdlib.h>
#include "structs.h"
#include "builtin.h"
#include "functions.h"
#include "environment.h"

TAGGED_VALUE *t = NULL;
TAGGED_VALUE *nil = NULL;

void init()
{
    t = alloc_symbol("t");
    nil = alloc_symbol("nil");

    /* add global symbols for true and false to the global environment (NULL) */
    add_binding(t, t, NULL);
    add_binding(nil, nil, NULL);

    /* add symbol bindings for all the built-in functions to the global
        environment (NULL) */

    /* all built-in functions handle thier parameters as a specially added list
        therefore, they all have their parameters set to NULL on lambda alloc */
    add_binding(alloc_symbol("atom?"),
        alloc_lambda(atomp, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("symbol?"),
        alloc_lambda(symbolp, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("eq?"),
        alloc_lambda(eqp, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("="),
        alloc_lambda(equal, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("<"),
        alloc_lambda(lessthan, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol(">"),
        alloc_lambda(greaterthan, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("<="),
        alloc_lambda(lessthanequal, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol(">="),
        alloc_lambda(greaterthanequal, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("+"),
        alloc_lambda(add, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("-"),
        alloc_lambda(sub, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("*"),
        alloc_lambda(prod, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("/"),
        alloc_lambda(divide, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("mod"),
        alloc_lambda(mod, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("expt"),
        alloc_lambda(expt, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("log"),
        alloc_lambda(log_num_base, 2, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("cons"),
        alloc_lambda(cons, 2, NULL, NULL, NULL, true), NULL);
```

```
    add_binding(alloc_symbol("first"),
        alloc_lambda(first, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("rest"),
        alloc_lambda(rest, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("read"),
        alloc_lambda(read_inp, 0, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("print"),
        alloc_lambda(print, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("explode"),
        alloc_lambda(explode, 1, NULL, NULL, NULL, true), NULL);
    add_binding(alloc_symbol("implode"),
        alloc_lambda(implode, 1, NULL, NULL, NULL, true), NULL);
}
```

8.2.11 memory.h

Garbage collector structures and function declarations.

```
#ifndef __MEMORY_H__
#define __MEMORY_H__

#include "structs.h"

typedef struct s_memory_ref {
    struct s_tagged_value *ref;
    struct s_memory_ref *next;
} MEMORY_REF;

TAGGED_VALUE *alloc_tv(TYPE type);
void gc(void);
void force_gc(void);
void gc_on_exit(void);

#endif /* __MEMORY_H__ */
```

8.2.12 memory.c

Garbage collector functions and utilities.

```
#include <stdio.h>
#include <stdlib.h>
#include "error.h"
#include "memory.h"
#include "structs.h"
#include "environment.h"

/* function decls */
static void mark(void);
static void markIt(TAGGED_VALUE *);
static void sweep(void);
static void free_tv(TAGGED_VALUE *);

extern ENV *current_env;         /* pointer to currently active environment,
    defined in env.c */

static MEMORY_REF *head = NULL;
static MEMORY_REF *last = NULL;

static long memory_ref_count = 0L;
static long max_memory_ref_before_gc = 64L;

/* create a new tagged value and keep track of it with a linked list of
    references */
TAGGED_VALUE *alloc_tv(TYPE type)
{
    /* alloc memory for tagged value */
    TAGGED_VALUE *new_tv = (TAGGED_VALUE *)malloc(sizeof(TAGGED_VALUE));
    if(NULL == new_tv)
    {
        fail("Unable to allocate memory for tagged value.");
    }
    new_tv->type = type;    /* set the value type */

    /* alloc memory for the reference to it */
    MEMORY_REF *new_ref = (MEMORY_REF *)malloc(sizeof(MEMORY_REF));
    if(NULL == new_ref)
    {
        fail("Unable to allocate memory for memory ref for gc.");
    }
    new_ref->ref = new_tv;  /* ref to new value */
    new_ref->next = NULL;   /* will be inserted at end of list */

    if(NULL == last)        /* is this the first item on the list */
    {
        head = new_ref;
        last = head;
    }
    else                    /* no, add it to the end then */
    {
        last->next = new_ref;
        last = new_ref;
    }

    memory_ref_count++;     /* keep track of the counts of objects */
```

```
    return new_tv;
}

void gc()
{
    /* check our current allocation stats before mark and sweep */
    if(memory_ref_count >= max_memory_ref_before_gc)
    {
        mark();
        sweep();

        max_memory_ref_before_gc <<= 1; /* double GC collection size */
    }
}

void force_gc()
{
    /* don't check current memory allocation, just do it */
    mark();
    sweep();
}

void gc_on_exit()
{
    /* no marking... just sweep everything.  we're exiting */
    sweep();
}

static void mark()
{
    /* start at the top of the call stack */
    ENV *env_ptr = current_env;
    while(env_ptr)              /* for each environment */
    {
        BINDING *current_bind = env_ptr->head; /* go through each binding */
        while(current_bind)
        {
            markIt(current_bind->symbol);      /* and mark its symbol and value
                */
            markIt(current_bind->value);

            current_bind = current_bind->next;
        }

        env_ptr = env_ptr->parent;
    }
}

static void markIt(TAGGED_VALUE *tv)
{
    if(tv->marked)      /* if it's already marked, skip it. prevents circular ref
        prob */
    {
        return;
    }
```

```
    tv->marked = 1;     /* mark the object for saving... it's special :) */
    if(CONS == tv->type)            /* if it's a cons cell */
    {
        markIt(tv->cell.car);     /* recursively mark the car */
        markIt(tv->cell.cdr);     /* and the cdr */
    }
    else if(LAMBDA == tv->type)     /* if it's a lambda */
    {
        if(!tv->lambda.is_builtin) /* built-in functions don't have an actual
            param list */
        {
            for(int i=0; i < tv->lambda.param_list.length; i++)
            {
                markIt(tv->lambda.param_list.params[i]);
            }
        }

        /* if the lambda is a closure, mark the symbols and values in those
            bindings too */
        BINDING *current_bind = tv->lambda.closure_bindings;
        while(current_bind)
        {
            markIt(current_bind->symbol);
            markIt(current_bind->value);

            current_bind = current_bind->next;
        }
    }
}

static void sweep()
{
    MEMORY_REF *prev=NULL, *current=head;
    MEMORY_REF *delete_mem_ref;

    /* for each item in the memory ref list... */
    while(current)
    {
        if(!current->ref->marked)       /* tagged value has not been marked */
        {
            if(NULL == prev)            /* prev hasn't moved yet, deleting first
                */
            {                                /* object in list */
                head = current->next;   /* just make sure head pointer is
                    adjusted */
            }
            else                        /* otherwise */
            {
                prev->next = current->next;  /* just update the previous object's
                    links */
            }

            free_tv(current->ref);      /* free the tagged value memory */
            delete_mem_ref = current;   /* pointer to associated memory ref to be
                freed */
            current = current->next;    /* advance current pointer down list,
```

```
                before */
            free(delete_mem_ref);           /* deleting the memory ref itself */
        }
        else                            /* value is marked, save it (no op) */
        {
            current->ref->marked = 0;   /* unmark for next mark/sweep round */
            prev = current;             /* advance previous pointer to current
                object */
            current = current->next;    /* advance current pointer down list */
        }
    }

    /* keep the last item in the list up to date */
    last = prev;
}

static void free_tv(TAGGED_VALUE *value)
{
    if(NULL == value)               /* nothing to delete */
    {
        return;
    }

    if(SYMBOL == value->type)       /* if it's a symbol in the tagged value */
    {                                   /* free the associated string memory */
        free(value->string_val);
    }
    else if(LAMBDA == value->type)
    {
        /* delete parameter list memory, if the lambda has params */
        free(value->lambda.param_list.params);

        /* delete closure environment (if it exists, might be NULL;) */
        BINDING *current_bind = value->lambda.closure_bindings;
        while(current_bind)
        {
            BINDING *bind_to_del = current_bind;
            current_bind = current_bind->next;
            free(bind_to_del);
        }
    }

    free(value);                    /* free tagged value memory */

    memory_ref_count--;             /* note deletion in the counts */
}
```

8.2.13 structs.h

Main data structures relating to tagged values.

```
#ifndef __STRUCTS_H__
#define __STRUCTS_H__

#include <stdbool.h>
#include "environment.h"

struct s_env;

typedef enum e_type {SYMBOL, INT, FLOAT, CONS, LAMBDA} TYPE;

typedef struct s_param_list {
    int length;
    struct s_tagged_value **params;
} PARAM_LIST;

typedef struct s_cell {
    struct s_tagged_value *car;
    struct s_tagged_value *cdr;
} CELL;

/* main data structure, the tagged value definition */
typedef struct s_tagged_value {
    unsigned char marked;           /* used for mark/sweep garbage collector */

    TYPE type;
    union {
        int int_val;
        double double_val;
        char *string_val;
        struct s_cell cell;
        struct {
            struct s_tagged_value *(*func_name)(struct s_tagged_value *this);
            struct s_binding *closure_bindings;
            bool is_builtin;
            PARAM_LIST param_list;
            struct s_tagged_value *parent_scope;
            struct s_env *running_env;
        } lambda;
    };
} TAGGED_VALUE;

TAGGED_VALUE *alloc_symbol(char *);
TAGGED_VALUE *alloc_int(int);
TAGGED_VALUE *alloc_double(double);
TAGGED_VALUE **alloc_params(int, ...);
BINDING *create_list_of_bindings(int, ...);
TAGGED_VALUE *alloc_lambda(TAGGED_VALUE *(*f)(TAGGED_VALUE*),int,
                           TAGGED_VALUE **, TAGGED_VALUE *, BINDING *, bool);
TAGGED_VALUE *construct(TAGGED_VALUE *, TAGGED_VALUE *);
TAGGED_VALUE *append(TAGGED_VALUE *, TAGGED_VALUE *);
TAGGED_VALUE *car(TAGGED_VALUE *);
TAGGED_VALUE *cdr(TAGGED_VALUE *);
int list_length(TAGGED_VALUE *);

#define IS_SYMBOL(s) (SYMBOL == s->type)
#define IS_LAMBDA(s) (LAMBDA == s->type)
```

```
#define IS_CONS(s) (CONS == s->type)
#define IS_INT(s) (INT == s->type)
#define IS_FLOAT(s) (FLOAT == s->type)
#define IS_NUMBER(s) (INT == s->type || FLOAT == s->type)
#define GET_NUM_VAL(s) ((INT == s->type) ? s->int_val : s->double_val)

#endif /* __STRUCTS_H__ */
```

8.2.14 structs.c

Source to tagged value utility and allocation functions.

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include "error.h"
#include "structs.h"
#include "memory.h"

extern TAGGED_VALUE *nil;
extern ENV *current_env;

TAGGED_VALUE *alloc_symbol(char *str)
{
    TAGGED_VALUE *new_tv = alloc_tv(SYMBOL);
    new_tv->string_val = (char *)malloc((strlen(str)*sizeof(char))+1);
    if(NULL == new_tv->string_val)
    {
        fail("Unable to allocate memory for symbol.");
    }
    strcpy(new_tv->string_val, str);

    return new_tv;
}

TAGGED_VALUE *alloc_int(int val)
{
    TAGGED_VALUE *new_tv = alloc_tv(INT);
    new_tv->int_val = val;

    return new_tv;
}

TAGGED_VALUE *alloc_double(double val)
{
    TAGGED_VALUE *new_tv = alloc_tv(FLOAT);
    new_tv->double_val = val;

    return new_tv;
}

TAGGED_VALUE *alloc_lambda(TAGGED_VALUE *(*f)(TAGGED_VALUE *),
    int param_length, TAGGED_VALUE **params, TAGGED_VALUE *parent_scope,
    BINDING *closure_bindings, bool is_builtin)
{
    TAGGED_VALUE *new_tv = alloc_tv(LAMBDA);
    new_tv->lambda.func_name = f;
    new_tv->lambda.param_list.length = param_length;
    new_tv->lambda.param_list.params = params;
    new_tv->lambda.closure_bindings = closure_bindings;
    new_tv->lambda.parent_scope = parent_scope;
    new_tv->lambda.is_builtin = is_builtin;
    new_tv->lambda.running_env = NULL;

    return new_tv;
}
```

```
TAGGED_VALUE **alloc_params(int n, ...)
{
    if(0 == n)
    {
        return NULL;
    }

    TAGGED_VALUE **params = (TAGGED_VALUE **)malloc(sizeof(TAGGED_VALUE *) * n);
    if(NULL == params)
    {
        fail("Cannot allocate parameters for lambda.");
    }

    va_list args;
    va_start(args, n);
    for(int i = 0; i < n; i++)
    {
        char *param_sym = va_arg(args, char *);
        params[i] = alloc_symbol(param_sym);
    }
    va_end(args);

    return params;
}

BINDING *create_list_of_bindings(int n, ...)
{
    if(0 == n)
    {
        return NULL;
    }

    BINDING *head = NULL;

    va_list args;
    va_start(args, n);
    for(int i = 0; i < n; i++)
    {
        BINDING *current;
        BINDING *b = va_arg(args, BINDING *);
        BINDING *new_bind = alloc_binding(b->symbol, b->value);
        if(0 == i)
        {
            head = new_bind;
        }
        else
        {
            current->next = new_bind;
        }

        current = new_bind;
    }
    va_end(args);

    return head;
}
```

```
TAGGED_VALUE *construct(TAGGED_VALUE *car, TAGGED_VALUE *cdr)
{
    TAGGED_VALUE *new_tv = alloc_tv(CONS);
    new_tv->cell.car = car;
    new_tv->cell.cdr = cdr;

    return new_tv;
}

TAGGED_VALUE *append(TAGGED_VALUE *value, TAGGED_VALUE *list)
{
    TAGGED_VALUE *new_tv = alloc_tv(CONS);
    new_tv->cell.car = value;
    new_tv->cell.cdr = nil;

    while(list->cell.cdr != nil)
    {
        list = list->cell.cdr;
    }

    list->cell.cdr = new_tv;

    return list;
}

TAGGED_VALUE *car(TAGGED_VALUE *list)
{
    if(CONS != list->type && nil != list)
    {
        fail("Cannot take car of something that is not a cons cell.");
    }

    if(nil == list)
        return nil;
    else
        return list->cell.car;
}

TAGGED_VALUE *cdr(TAGGED_VALUE *list)
{
    if(CONS != list->type && nil != list)
    {
        fail("Cannot take cdr of something that is not a cons cell.");
    }

    if(nil == list)
        return nil;
    else
        return list->cell.cdr;
}
```



```
int list_length(TAGGED_VALUE *list)
{
    if(CONS != list->type && nil != list)
    {
```

```
            fail("Cannot get length of something that is not a cons cell.");
        }

        int len = 0;

        while(nil != list)
        {
            len++;
            list = list->cell.cdr;
        }

        return len;
    }
```

References

[1] Harold Abelson, Gerald Jay Sussman, and Juile Sussman. *Structure and Interpretation of Computer Programs*. 2nd. Cambridge MA: The MIT Press, 1996. ISBN: 0262510871.

[2] Alexander Burger. *The PicoLisp Reference*. URL: `http://software-lab.de/doc/ref.html`.

[3] Paul Graham. *The Roots of Lisp*. May 2001. URL: `http://www.paulgraham.com/rootsoflisp.html`.

[4] John McCarthy. *LISP 1.5 Programmer's Manual*. 2nd. Cambridge MA: The MIT Press, 1962. ISBN: 0262130114.