Build Your Own Lisp

Introduction

About

In this book you'll learn the C programming language, and at the same time learn how to build *your very own programming language*, a minimal Lisp, in under 1000 lines of code! We'll be using a library to do some of the initial work, so I'm cheating a little bit on the line count, but the rest of the code will be completely original, and you really will create a powerful little Lisp by the end.

This book is somewhat inspired by online tutorials such as Write Yourself a Scheme in 48 Hours which go through the steps of building a programming language from scratch. I wrote this book to show that this kind of fun and creative project is a great way to learn a language, and not limited to abstract high-level languages, or experienced programmers.

Many people are keen to learn C, but have nowhere to start. Well here is your excuse. If you follow this book I can promise that, in the worst case, you'll get a cool new programming language to play with, and hopefully you'll become an experienced C programmer too!

Who this is for

This book is for anyone wanting to learn C, or who has once wondered how to build their own programming language. This book is not suitable as a first programming language book, but anyone with at least some minimal programming experience, in any language, should find something new and interesting inside.

I've tried to make this book as friendly as possible to beginners. I welcome beginners the most because they have so much to discover! But beginners will also find this book hard. We will be covering lots of new concepts, and essentially learning two new programming languages at once.

If you look for help you may find people are not patient with you. You may find that, rather than help, they take the time to express how much *they* know about the subject. Esteemed people with lots of experience might tell you that you are wrong. The subtext to their tone might be that you should stop now, rather than inflict your bad code on the world.

After a couple of engagements like this you may decide that you are *not a programmer*, or *don't really like programming*, or that you just *don't get it*. You may have thought that you *once enjoyed* the idea of building your own programming language, but now you have realized that it is too abstract and you *don't care any more*. Now you are concerned with your other passion, and any insight that may have been playful, joyful or interesting will now have become an obstacle.

For this I can only apologize. Programmers can be hostile, macho, arrogant, insecure, and aggressive. There is no excuse for this behaviour. Know that I am on your side. No one *gets it* at first. Everyone struggles and doubts their

abilities. Please don't give up or let the joy be sucked out of the creative experience. Be proud of what you create no matter what it is. People like me don't want you to stop programming. We want to hear your voice, and what you have to say, even if you do not shout as loud as others.

Why learn C

C is one of the most popular and influential programming languages in the world. It is the language of choice for development on Linux, and has been used extensively in the creation of OS X and to some extent Microsoft Windows. It is used on micro-computers too. Your fridge and car probably run on it. In modern software development, the use of C may be escapable, but its legacy is not. Anyone wanting to make a career out of software development would be smart to learn C.

But C is not about software development and careers. C is about **freedom**. It rose to fame on the back of technologies of collaboration and freedom - Unix, Linux, and The Libre Software Movement. It personifies the idea of personal liberty within computing. It wills you to take control of the technology affecting your life.

In this day and age, when technology is more powerful than ever, this could not be more important.

The ideology of freedom is reflected in the nature of C itself. There is little C hides from you, including its warts and flaws. There is little C stops you from doing, including breaking your programs in horrible ways. When programming in C you do not stand on a path, but a plane of decision, and C dares you to decide what to do.

C is also the language of fun and learning. Before the mainstream media got hold of it we had a word for this. *Hacking*. The philosophy that glorifies what is fun and clever. Nothing to do with the illegal unauthorised access of other peoples' computers. Hacking is the philosophy of exploration, personal expression, pushing boundaries, and breaking the rules. It stands against hierarchy and bureaucracy. It celebrates the individual. Hacking baits you with fun, learning, and glory. Hacking is the promise that with a computer and access to the internet, you have the agency to change the world.

To want to master C, is to care about what is powerful, fun, clever, and free. To become a programmer with all the vast powers of technology at his or her fingertips. And the responsibility to say *"I will do good with this"*.

How to learn C

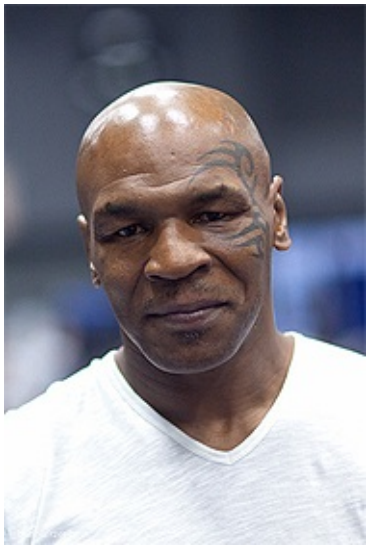
There is no way around the fact that C is a difficult language. It has many concepts that are unfamiliar, and it makes no attempts to help a new user. In this book I am *not* going to cover in detail things like the syntax of the language, or how to write loops and conditional statements.

I will, on the other hand, show you how to build a *real world* program in C. This approach is always more difficult for the reader, but hopefully will teach you many implicit things a traditional approach cannot. I can't make any promise that this book will make you a confident user of C. What I can promise, is that those 1000 lines of code are going to be packed with content - and you will learn *something* worthwhile.

This book consists of 16 short chapters. How you complete these is up to you. It may well be possible to blast through this book over a weekend, or to take it more slowly, and do a chapter or two each evening over a week. It shouldn't take very long to complete, and will hopefully leave you with a taste for developing your language further.

Why build a Lisp

The language we are going to be building in this book is a Lisp. This is a family of programming languages characterized by the fact that all their computation is represented by *lists*. This may sound scarier than it is. Lisps are actually very easy, distinctive, and powerful languages.



Building a Lisp is a great project for so many reasons. It puts you in the shoes of language designers, and gives you an appreciation for the whole process of programming, from language all the way down to machine. It teaches you about functional programming, and different ways to view computation out of the norm. The final product you are rewarded with provides a template for future thoughts and developments, giving you a starting ground for trying new things. It simply isn't possible to comprehend the creativity and cleverness that goes into programming and computer science until you explore languages themselves.

The type of Lisp we'll be building is one I've invented for the purposes of this book. I've designed it for minimalism, simplicity and clarity, and I've become quite fond of it along the way. I hope you come to like it too. Conceptually, syntactically, and in implementation this brand of Lisp has a number of serious differences to other major brands of Lisp. So much so that I'm sure I will be getting e-mails from Lisp programmers telling me it *isn't a Lisp* because it *doesn't do/have/look-like this or that*.

I've not made this Lisp different to confuse beginners or to spread untruths. I've made it different because different is good.

If you are looking to learn about the semantics and behaviours of conventional Lisps, and how to program them, this book may not be for you. What this book offers instead is new and unique concepts, self expression, creativity, and fun. Whatever your motivation, heed this disclaimer now. Not everything I say will be objectively correct or true! You will have to decide that for yourselves.

Your own Lisp

The best way to follow this book is to, as the title says, write *your own* Lisp. If you are feeling confident enough I want you to add your own features, modifications and changes. Your Lisp should suit you and your own philosophy on what is right or true. Throughout the book I'll be giving description and insight, but with it I'll be providing *a lot* of code. This will make it easy to follow along by copy and pasting each section into your program without really understanding. *Please do not do this!*.

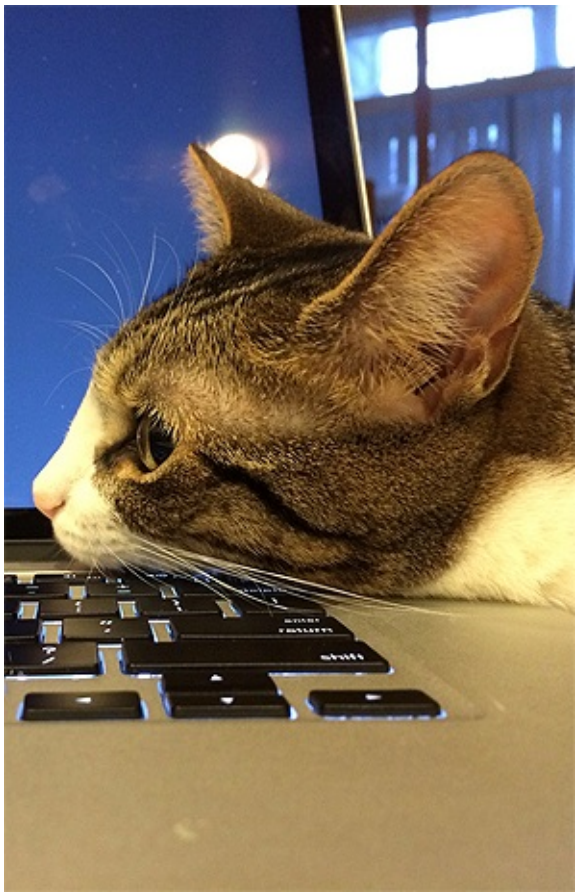
Type out each piece of sample code yourself. This is called *The Hard Way*. Not because it is hard technically, but

because it requires discipline. By doing things *The Hard Way* you will come to understand the reasoning behind what you are typing. Ideally things will click as you follow it along character by character. When *reading* you may have an intuition as to why it *looks* right, or what *may* be going on, but this will not always translate to a real understanding unless you do the *writing* yourself!

In a perfect world you would use my code as a reference - an instruction booklet and guide as to building the programming language you always dreamed of. In reality this isn't practical or viable. But the base philosophy remains. If you want to change something, do it.

Installation

Setup



Before we can start programming in C we'll need to install a couple of things, and set up our environment so that we have everything we need. Because C is such a universal language this should hopefully be fairly simple. Essentially we need to install two main things. A *text editor* and a *compiler*.

Text Editor

A text editor is a program that allows you to edit text files in a way suitable for programming.

On **Linux** the text editor I recommend is gedit. Whatever other basic text editor comes installed with your distribution will also work well. If you are a Vim or Emacs user these are fine to use. Please don't use an IDE. It isn't required for such a small project and won't help in understanding what is going on.

On **Mac** A simple text editor that can be used is TextWrangler. If you have a different preference this is fine, but please don't use XCode for text editing. This is a small project and using an IDE won't help you understand what is going on.

On **Windows** my text editor of choice is Notepad++. If you have another preference this is fine. Please *don't* use *Visual Studio* as it does not have proper support for C programming. It you attempt to use it you will run into many problems.

Compiler

The compiler is a program that transforms the C source code into a program your computer can run. The installation process for these is different depending on what operating system you are running.

Compiling and running C programs is also going to require really basic usage of the command line. This I will not cover,

so I am going to assume you have at least some familiarity with using the command line. If you are are worried about this then search online for information on using it, relevant to your operating system.

On **Linux** you can install a compiler by downloading some packages. If you are running Ubuntu or Debian you can install everything you need with the following command `sudo apt-get install build-essential`. If you are running Fedora or a similar Linux variant you can use this command `su -c "yum groupinstall development-tools"`.

On **Mac** you can install a compiler by downloading and installing the latest version of XCode from Apple. If you are unsure of how to do this you can search online for "installing xcode" and follow any advice shown. You will then need to install the *Command Line Tools*. On Mac OS X 10.9 this can be done by running the command `xcode-select --install` from the command line. On versions of Mac OS X prior to 10.9 this can be done by going to XCode Preferences, Downloads, and selecting *Command Line Tools* for Installation.

On **Windows** you can install a compiler by downloading and installing MinGW. If you use the installer at some point it may present you with a list of possible packages. Make sure you pick at least `mingw32-base` and `msys-base`. Once installed you need to add the compiler and other programs to your system `PATH` variable. To do this follow these instructions appending the directory `;C:\MinGW\bin` to the variable called `PATH`. You can create this variable if it doesn't exist. You may need to restart `cmd.exe` for the changes to take effect. This will allow you to run a compiler from the command line `cmd.exe`. It will also install other programs which make `cmd.exe` act like a Unix command line.

Testing the Compiler

To test if your C compiler is installed correctly type the following into the command line.

```
cc --version
```

If you get some information about the compiler version echoed back then it should be installed correctly. You are ready to go! If you get any sort of error about an unrecognised or not found command, then it is not ready. You may need to restart the command line or your computer for changes to take effect.

Hello World

Now that your environment is set up, start by opening your text editor and inputting the following program. Create a directory where you are going to put your work for this book, and save this file as `hello_world.c`. This is your first C program!

```
#include <stdio.h>

int main(int argc, char** argv) {
  puts("Hello, world!");
  return 0;
}
```

This may look like a lot of crazy symbols that make very little sense. I'll try to explain it step by step.

In the first line we *include* what is called a *header*. This statement allows us to use the functions from `stdio.h`, the standard input and output library which comes included with C. One of the functions from this library is the `puts` function you see later on in the program.

Next we *declare* a function called `main`. This function is declared to output an `int`, and take as input an `int` called `argc` and a `char**` called `argv`. All C programs must contain this function. All programs start running from this function.

Inside `main` the `puts` function is *called* with the argument `"Hello, world!"`. This outputs the message `Hello, world!` to the command line. The function `puts` is short for *put string*. The second statement inside the function is `return 0;`. This tells the `main` function to finish and return `0`. When a C program returns `0` this indicates there have been no errors running the program.

Compilation

Before we can run this program we need to compile it. This will produce the actual *executable* we can run on our computer. Open up the command line and browse to the directory that `hello_world.c` is saved in. You can then compile your program using the following command.

```
cc -std=c99 -Wall hello_world.c -o hello_world
```

This compiles the code in `hello_world.c`, reporting any warnings, and outputs the program to a new file called `hello_world`. We use the `-std=c99` flag to tell the compiler which *version* or *standard* of C we are programming with. This lets the compiler ensure our code is standardized, so that people with different operating systems or compilers will be able to use our code.

If successful you should see the output file in the current directory. This can be run by typing `./hello_world` (or just `hello_world` on Windows). If everything is correct you should see a friendly `Hello, world!` message appear.

Congratulations! You've just compiled and run your first C program.

Errors

If there are some problems with your C program the compilation process may fail. These issues can range from simple syntax errors, to other complicated problems that are hard to understand.

Sometimes the error message from the compiler will make sense, but if you are having trouble understanding it try searching online for it. You should see if you can find a concise explanation of what it means, and work out how to correct it. Remember this: there are many people before you who have struggled with exactly the same problems.

Sometimes there will be many compiler errors stemming from one source. Always work through compiler errors from first to last.

Sometimes the compiler will compile a program, but when you run it it will crash. Debugging C programs in this situation is hard. It can be an art far beyond the scope of this book.

My first port of call for debugging a crashing C program is to print out lots of information as the program is running. Using this method I can try to isolate exactly what part of the code is incorrect and what, if anything, is going wrong up until the crash. For beginners I would recommend this technique. It is a debugging technique which is *active*. This is the important thing. As long as you are doing *something*, and not just staring at the code, the process is less painful and the temptation to give up is lessened.

For people feeling more confident a program called `gdb` can be used to debug your C programs. This can be difficult and complicated to use, but it is also very powerful and can give you extremely valuable information and what went wrong and where. Information on how to use `gdb` can be found online.

On **Mac** the most recent versions of OS X don't come with `gdb`. Instead you can use `lldb` which does largely the same

job.

On **Linux** or **Mac** `valgrind` can be used to aid the debugging of memory leaks and other more nasty errors. Valgrind is a tool that can save you hours, or even days, of debugging. It does not take much to get proficient at it, so investigating it is highly recommended. Information on how to use it can be found online.

Documentation

Through this book you may come across functions in some example code that you don't recognize. You might wonder what it does. In this case you will want to look toward the online documentation of the standard library. This will explain all the functions included in the standard library, what they do, and how to use them.

Reference

What is this section for? In this section I'll link to the code I've written for this particular chapter of the book. When finishing with a chapter your code should probably look similar to mine. This code can be used for reference if the explanation has been unclear. If you encounter a bug please do not copy and paste my code into your project. Try to track down the bug yourself and use my code as a reference to highlight what may be wrong, or where the error may lie.
hello_world.c

Bonus Marks

What is this section for? In this section I'll list some things to try for fun, and learning. It is good if you can attempt to do some of these challenges. Some will be easy, while some will be very difficult. For this reason don't worry if you can't figure them all out. Some might not even be possible! Many will require some research on the internet. This is an integral part of learning a new language so should not be avoided. The ability to teach yourself things is one of the most valuable skills in programming. See how many you can complete for each chapter. Move on when you get bored.

- › Change the `Hello World!` greeting given by your program to something different.
- › What happens when no `main` function is given?
- › Use the online documentation to lookup the `puts` function.
- › Look up how to use `gdb` and run it with your program.

Basics

Overview

Mucha : Couverture du programme pour sa soirée du 6 novembre 1897.

In this chapter I've prepared a quick overview of the basic features of C. There are very few *features* in C, and the syntax is relatively simple. But this doesn't mean it is easy. All the depth hides below the surface. Because of this we're going to cover the *features* and *syntax* fairly quickly now, and see them in greater depth as we continue.

The goal of this chapter is get everyone on the same page. People totally new to C should therefore read it in some consideration, and take some time over it, while those with some existing experience may find it easier to skim and return to later as required.

Programs

A program in C consists of only *function definitions* and *structure definitions*.

Therefore a source file is simply a list of *functions* and *types*. These functions can call each other or themselves, and can use any data types that have been declared or are built into the language.

It is possible to call functions in other libraries, or to use their data types. This is how layers of complexity are accumulated in C programming.

As we saw in the previous chapter, the execution of a C program always starts in the function called `main`. From here it calls more and more functions, to perform all the actions it requires.

Variables

Functions in C consists of manipulating *Variables*. These are items of data which we give a name to.

Every variable in C has an explicit *type*. These types are declared by ourselves or built into the language. We can declare a new variable by writing the name of its type, followed by its name, and optionally setting it to some value using `=`. This

declaration is called a *statement*, and we terminate all *statements* in C with a semicolon `;`.

To create a new `int` called `count` we could write the following...

```
int count;
```

Or to declare it and set the value...

```
int count = 10;
```

Here are some descriptions and examples of some of the built in types.

`void`	Empty Type	
`char`	Single Character/Byte	`char last_initial = 'H';`
`int`	Integer	`int age = 23;`
`long`	Integer that can hold larger values	`long age_of_universe = 13798000000;`
`float`	Decimal Number	`float liters_per_pint = 0.568f;`
`double`	Decimal Number with more precision	`double speed_of_swallow = 0.01072896;`

Function Declarations

A Function is a computation that manipulates variables, and optionally changes the state of the program. It takes as input some variables and returns some single variable as output.

To declare a function we write the type of the variable it returns, the name of the function, and then in parenthesis a list of the variables it takes as input , separated by commas. The contents of the function is put inside curly brackets `{}`, and lists all of the statements the function executes, terminated by semicolons `;`. A `return` statement is used to let the function finish and output a variable.

For example a function that takes two `int` variables called `x` and `y` and adds them together could look like this.

```
int add_together(int x, int y) {
  int result = x + y;
  return result;
}
```

We call functions by writting their name and putting the arguments to the function in parenthesis, separated by commas. For example to call the above function and store the result in a variable `added` we would write the following.

```
int added = add_together(10, 18);
```

Structure Declarations

Structures are used to declare new types*. Structures are several variables bundled together into a single package.

We can use structure to represent more complex data types. For example to represent a point in 2D space we could create a structure called `point` that packs together two `float` (decimal) values called `x` and `y`. To declare structures we can use the `struct` keyword in conjunction with the `typedef` keyword. Our declaration would look like this.

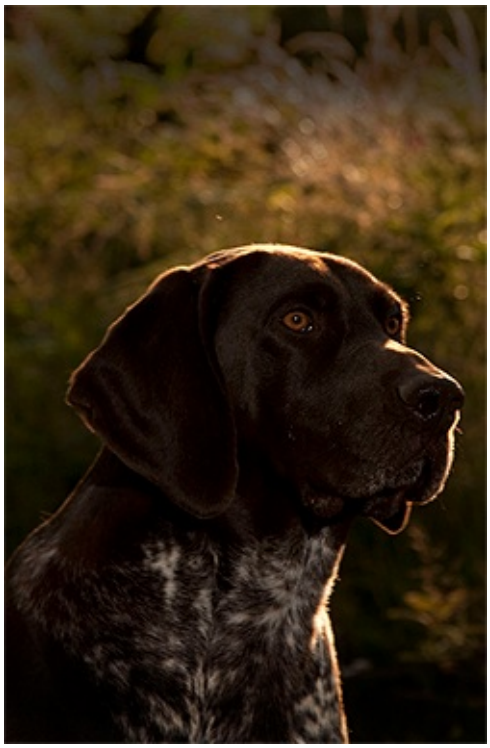
```
typedef struct {
  float x;
  float y;
} point;
```

We should place this definition above any functions that wish to use it. This type is no different to the built in types, and we can use it in all the same ways. To access an individual field we use a dot `.`, followed by the name of the field, such as `x`.

```
point p;
p.x = 0.1;
p.y = 10.0;

float length = sqrt(p.x * p.x + p.y * p.y);
```

Pointers



A pointer is a variation on a normal type where the type name is suffixed with an asterisk. For example we could declare a *pointer to an integer* by writing `int*`. We already saw a pointer type `char** argv`. This is a *pointer to pointers to characters*, and is used as input to `main` function.

Pointers are used for a whole number of different things such as for strings or lists. These are a difficult part of C and will be explained in much greater detail in later chapters. We won't make use of them for a while, so for now it is good to simply know they exist, and how to spot them. Don't let them scare you off!

Strings

In C strings are represented by the pointer type `char*`. Under the hood they are stored as a list of characters, where the final character is a special character called the *null terminator*. Strings are a complicated, and important part of C, which we'll learn to use effectively in the next few chapters.

Strings can also be declared literally by putting text between quotation marks. We used this in the previous chapter with our string `"Hello, World!"`. For now, remember that if you see `char*`, you can read it as a *string*.

Conditionals

Conditional statements let the program perform some code only if certain conditions are met.

To perform code under some condition we use the `if` statement. This is written as `if` followed by some condition in parenthesis, followed by the code to execute in curly brackets. An `if` statement can be followed by an optional `else` statement, followed by other statements in curly brackets. The code in these brackets will be performed in the case the conditional is false.

We can test for multiple conditions using the logical operators `||` for *or*, and `&&` for *and*.

Inside a conditional statement's parenthesis any value that is not `0` will evaluate to true. This is important to remember as many conditions use this to check things implicitly.

If we wished to check if an `int` called `x` was greater than `10` and less then `100`, we would write the following.

```
if (x > 10 && x < 100) {
  puts("x is greater than ten and less than one hundred!");
} else {
  puts("x is either less than eleven or greater than ninety-nine!");
}
```

Loops

Loops allow for some code to be repeated until some condition becomes false, or some counter elapses.

There are two main loops in C. The first is a `while` loop. This loop repeatedly executes a block of code until some condition becomes false. It is written as `while` followed by some condition in parenthesis, followed by the code to execute in curly brackets. For example a loop that counts downward from `10` to `1` could be written as follows.

```
int i = 10;
while (i > 0) {
  puts("Loop Iteration");
  i = i - 1;
}
```

The second kind of loop is a `for` loop. Rather than a condition, this loop requires three expressions separated by semicolons `;`. These are an *initialiser*, a *condition* and an *incrementer*. The *initialiser* is performed before the loop starts, the *condition* is checked at the end of each iteration of the loop, and if false the loop is exited. The *incrementer* is performed before the next iteration of the loop. These loops are often used for counting as they are more compact than the `while` loop.

For example to write a loop that counts up from `0` to `9` we might write the following. In this case the `++` operator increments the variable `i`.

```
for (int i = 0; i < 10; i++) {
  puts("Loop Iteration");
}
```

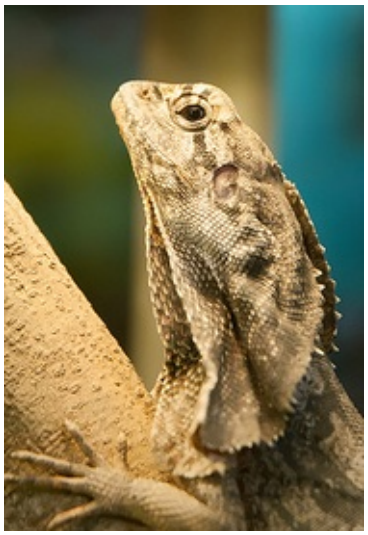
Bonus Marks

- › Use a `for` loop to print out `Hello World!` five times.
- › Use a `while` loop to print out `Hello World!` five times.
- › Declare a function that outputs `Hello World!` `n` number of times. Call this from `main`.
- › What built in types are there other than the ones listed?

- › What other conditional operators are there other than *greater than* `>`, and *less than* `<`?
- › What other mathematical operators are there other than *add* `+`, and *subtract* `-`?
- › What is the `+=` operator, and how does it work?
- › What is the `do` loop, and how does it work?
- › What is the `switch` statement and how does it work?
- › What is the `break` keyword and what does it do?
- › What is the `continue` keyword and what does it do?
- › What does the `typedef` keyword do exactly?

An Interactive Prompt

Read, Evaluate, Print



As we build our programming language we'll need some way to interact with it. C uses a compiler, where you can change the program, recompile and run it. It'd be good if we could do something better, and interact with the language dynamically. Then we test its behaviour under a number of conditions very quickly. For this we can built something called an *interactive prompt*.

This is a program that prompts the user for some input, and when supplied with it, replies back with some message. Using this will be the easiest way to test our programming language and see how it acts. This system is also called a *REPL*, which stands for *read-evaluate-print loop*. It is a common way of interacting with a programming language which you may have used before in languages such as *Python*.

Before building a full *REPL* we'll start with something simpler. We are going to make a system that prompts the user, and echoes any input straight back. If we make this we can later extend it to parse the user input and evaluate it, as if it were an actual Lisp program.

An Interactive Prompt

For the basic setup we want to write a loop which repeatedly writes out a message, and then waits for some input. To get user input we can use a function called `fgets`, which reads any input up until a newline. We need somewhere to store this user input. For this we can declare a constantly sized input buffer.

Once we have this user input stored we can then print it back to the user using a function called `printf`.

```c
#include <stdio.h>

/* Declare a static buffer for user input of maximum size 2048 */
static char input[2048];

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

  /* Print Version and Exit Information */
  puts("Lispy Version 0.0.0.0.1");
  puts("Press Ctrl+c to Exit\n");

  /* In a never ending loop */
  while (1) {

    /* Output our prompt */
    fputs("lispy> ", stdout);

    /* Read a line of user input of maximum size 2048 */
    fgets(input, 2048, stdin);

    /* Echo input back to user */
    printf("No you're a %s", input);
  }

  return 0;
}
```

What is that text in light gray? The above code contains *comments*. These are sections of the code between `/*` `*/` symbols, which are ignored by the compiler, but are used to inform the person reading what is going on. Take notice of them!
Lets go over this program in a little more depth.

The line `static char input[2048];` declares a global array of 2048 characters. This is a reserved block of data we can access anywhere from our program. In it we are going to store the user input which is typed into the command line. The `static` keyword make this variable local to this file, and the `[2048]` section is what declares the size.

We write an infinite loop using `while (1)`. In a conditional block `1` always evaluates to true. Therefore commands inside this loop will run forever.

To output our prompt we use the function `fputs`. This is a slight variation on `puts` which does not append a newline character. We use the `fgets` function for getting user input from the command line. Both of these functions require some file to write to, or read from. For this we supply the special variables `stdin` and `stdout`. These are declared in `<stdio.h>` and are special file variables representing input to, and output from, the command line. When passed this variable the `fgets` function will wait for a user to input a line of text, and when it has it will store it into the `input` buffer, including the newline character. So that `fgets` does not read in too much data we also must supply the size of the buffer `2048`.

To echo the message back to the user we use the function `printf`. This is a function that provides a way of printing messages consisting of several elements. It matches arguments to patterns in the given string. For example in our case we can see the `%s` pattern in the given string. This means that it will be replaced by whatever argument is passed in next, interpreted as a string.

For more information on these different patterns please see the documentation on `printf`.

How am I meant to know about functions like `fgets` and `printf`? It isn't immediately obvious how to know about these standard functions, and when to use them. When faced with a problem it takes experience to know when it has been solved for you by library functions. Luckily C has a very small standard library and almost all of it can be learnt in practice. If you want to do something that seems quite basic, or fundamental, it is worth looking at the reference documentation for the standard library and checking if there are any functions included that do what you want.

Compilation

You can compile this with the same command as was used in the second chapter.

```
cc -std=c99 -Wall prompt.c -o prompt
```

After compiling this you should try to run it. You can use `Ctrl+c` to quit the program when you are done. If everything is correct your program should run something like this.

```
Lispy Version 0.0.0.0.1
Press Ctrl+c to Exit

lispy> hello
No You're a hello
lispy> my name is Dan
No You're a my name is Dan
lispy> Stop being so rude!
No You're a Stop being so rude!
lispy>
```

Editing input

If you're working on Linux or Mac you'll notice some weird behaviour when you use the arrow keys to attempt to edit your input.

```
Lispy Version 0.0.0.0.3
Press Ctrl+c to Exit

lispy> hel^[[D^[[C
```

Using the arrow keys is creating these weird characters `^[[D` or `^[[C`, rather than moving the cursor around in the input. What we really want is to be able to move around on the line, deleting and editing the input in case we make a mistake.

On Windows this behaviour is the default. On Linux and Mac it is provided by a library called `editline`. On Linux and Mac we need to replace our calls to `fputs` and `fgets` with calls to functions this library provides.

If you're developing on Windows and just want to get going, feel free to skip to the end of this chapter as the next few sections may not be relevant.

Using Editline

The library `editline` provides two functions we are going to use called `readline` and `add_history`. This first function, `readline` is used to read input from some prompt, while allowing for editing of that input. The second function `add_history` lets us record the history of inputs so that they can be retrieved with the up and down arrows.

We replace `fputs` and `fgets` with calls to these functions to get the following.

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

#include <editline/readline.h>
#include <editline/history.h>

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

  /* Print Version and Exit Information */
  puts("Lispy Version 0.0.0.0.1");
  puts("Press Ctrl+c to Exit\n");

  /* In a never ending loop */
  while (1) {

    /* Output our prompt and get input */
    char* input = readline("lispy> ");

    /* Add input to history */
    add_history(input);

    /* Echo input back to user */
    printf("No you're a %s\n", input);

    /* Free retrived input */
    free(input);

  }

  return 0;
}
```

We have *included* a few new *headers*. There is `#include <stdlib.h>`, which gives us access to the `free` function used later on in the code. We have also added `#include <editline/readline.h>` and `#include <editline/history.h>` which give us access to the `editline` functions, `readline` and `add_history`.

Instead of prompting, and getting input with `fgets`, we do it in one go using `readline`. The result of this we pass to `add_history` to record it. Finally we print it out as before using `printf`.

Unlike `fgets`, the `readline` function strips the trailing newline character from the input, so we need to add this to our `printf` function. We also need to delete the input given to us by the `readline` function using `free`. This is because unlike `fgets`, which writes to some existing buffer, the `readline` function allocates new memory when it is called. When to free memory is something we cover in depth in later chapters.

Compiling with Editline

If you try to compile this right away with the previous command you'll get an error. This is because you first need to install the `editline` library on your computer.

```
fatal error: editline/readline.h: No such file or directory #include <editline/readline.h>
```

On **Linux** this can be done using the command `sudo apt-get install libedit-dev`. On Fedora or similar you can use the command `su -c "yum install libedit-dev*"`

On **Mac** the `editline` library should have been installed alongside the *Command Line Tools*. If you get an error about the history header not being found you can either try removing that line of code or installing the `readline` library, which can be used as a drop-in replacement. This can be installed using Homebrew or MacPorts.

Once you have installed this you can try to compile it again. This time you'll get a different error.

```
undefined reference to `readline'
undefined reference to `add_history'
```

This means that you haven't *linked* your program to `editline`. This *linking* process allows the compiler to directly embed calls to `editline` in your program. You can make it link by adding the flag `-ledit` to your compile command, just before the output flag.

```
cc -std=c99 -Wall prompt.c -ledit -o prompt
```

Hopefully now you should be able to *compile* and *link* your program with `editline`. Run it and check that now you can edit inputs as you type them in.

It's still not working! Some systems might have slight variations on how to install, include, and link to `editline`. For example on Mac and some other systems the history header may not be required and so that line of code can be removed. On Arch linux the editline history header is `histedit.h`. If you are having trouble search online and see if you can find distribution specific instructions on how to use `editline` or `readline`, an equivalent library.

The C Preprocessor

For such a small project it might be okay that we have to program differently depending on what operating system we are using, but if I want to send my source code to a friend on different operating system to give me a hand with the programming, it is going to cause problem. In an ideal world I'd wish for my source code to be able to compile no matter where, or on what computer, it is being compiled. This is a general problem in C, and it is called *portability*. There is not always an easy or correct solution.

But C does provide a mechanism to help, called *the preprocessor*.

The preprocessor is a program that runs before the compiler. It has a number of purposes, and we've been actually using it already without knowing. Any line that starts with a octothorpe `#` character (hash to you and me) is a preprocessor command. We've been using it to *include* header files, giving us access to functions from the standard library and others.

Another use of the preprocessor is to detect which operating system the code is being compiled on, and to use this to emit different code.

This is exactly how we are going to use it. If we are running Windows we're going to let the preprocessor emit code with some fake `readline` and `add_history` functions I've prepared, otherwise we are going to include the headers from `editline` and use these.

To declare what code the compiler should emit we can wrap it in `#ifdef`, `#else`, and `#endif` preprocessor statements. These are like an `if` function that happens before the code is compiled. All the contents of the file from the first `#ifdef` to the next `#else` are used if the condition is true, otherwise all the contents from the `#else` to the final `#endif` are used instead. By putting these around our fake functions, and our editline headers, the code that is emitted should compile on Windows, Linux or Mac!

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

/* If we are compiling on Windows compile these functions */
#ifdef _WIN32

#include <string.h>

static char buffer[2048];

/* Fake readline function */
char* readline(char* prompt) {
  fputs(prompt, stdout);
  fgets(buffer, 2048, stdin);
  char* cpy = malloc(strlen(buffer)+1);
  strcpy(cpy, buffer);
  cpy[strlen(cpy)-1] = '\0';
  return cpy;
}

/* Fake add_history function */
void add_history(char* unused) {}

/* Otherwise include the editline headers */
#else

#include <editline/readline.h>
#include <editline/history.h>

#endif

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

  puts("Lispy Version 0.0.0.0.1");
  puts("Press Ctrl+c to Exit\n");

  while (1) {

    /* Now in either case readline will be correctly defined */
    char* input = readline("lispy> ");
    add_history(input);

    printf("No you're a %s\n", input);
    free(input);

  }

  return 0;
}
```

Reference

prompt.c

Bonus Marks

- › Change the prompt from `lispy>` to something of your choice.
- › Change what is echoed back to the user.
- › Add an extra message to the *Version* and *Exit* Information.
- › What does the `\n` mean in those strings?
- › What other patterns can be used with `printf`.
- › What happens when you pass `printf` a variable does not match the pattern?
- › What does the preprocessor command `#ifndef` do?
- › What does the preprocessor command `#define` do?
- › If `_WIN32` is defined on windows, what is defined for Linux or Mac?

Languages

What is a Programming Language?

A programming language is very similar to a real language. There is a structure behind it, and some rules which dictate what is, and isn't, a valid thing to say. When we read and write natural language, we are unconsciously learning these rules, and the same is true for programming languages. We can utilise these rules to understand others, and generate our own speech, or code.

In the 1950s the linguist *Noam Chomsky* formalised a number of important observations about languages. Many of these form the basis of our understanding of language today. One of these was the observation that natural languages are built up of recursive and repeated substructures.

As an example of this, we can examine the phrase.

› `The cat walked on the carpet.`

Using the rules of English, the noun `cat` can be replaced by two nouns separated by `and`.

› `The` **`cat and dog`** `walked on the carpet.`

Each of these new nouns could in turn be replaced again. We could use the same rule as before, and replace `cat` with two new nouns joined with `and`. Or we could use a different rule and replace each of the nouns with an adjective and a noun, to add description to them.

› `The` **`cat and mouse`** `and dog walked on the carpet.`

› `The` **`white cat`** `and` **`black dog`** `walked on the carpet.`

These are just two examples, but English has many different rules for how types of words can be changed, manipulated and replaced.

We notice this exact behaviour in programming languages too. In C, the body of an `if` statement contains a list of new statements. Each of these new statements, could themselves be another `if` statement. These repeated structures and replacements are reflected in all parts of the language. These are sometimes called *re-write rules* because they tell you how one thing can be *re-written* as something else.

› `if (x > 5) { return x; }`

› `if (x > 5) {` **`if (x > 10) { return x; }`** `}`

The consequence of this observation by *Noam Chomsky* was important. It meant that although there is an infinite number of different things that can be said, or written down, in a particular language; it is still possible to process and understand

all of them, with a finite number of these re-write rules. The name given to a set of these re-write rules is a *grammar*.

We can describe re-write rules in a number of ways. One way is textual. We could say something like, "a *sentence* must contain a *verb phrase*", or "a *verb phrase* can be either a *verb* or, an *adverb* and a *verb*". This method is good for humans but it is too vague for computers to understand. When programming we need to write down a more formal description of a grammar.

To write a programming language such as our Lisp we are going to need to understand grammars. For reading in the user input we need to write a *grammar* which describes it. Then we can use it along with our user input, to decide if the input is valid. We can also use it to build a structured internal representation, which will make the job of *understanding* it, and then *evaluating* it, performing the computations encoded within, much easier.

This is where a library called `mpc` comes in.

Parser Combinators

`mpc` is a *Parser Combinator* library written by yours truly. This means it is a library that allows you to build programs that understand and process particular languages. These are known as *parsers*. There are many different ways of building parsers, but the nice thing about using a *Parser Combinator* library is that it lets you build *parsers* easily, just by specifying the *grammar* ... sort of.

Many *Parser Combinator* libraries actually work by letting you write normal code that *looks a bit like* a grammar, not by actually specifying a grammar directly. In many situations this is fine, but sometimes it can get clunky and complicated. Luckily for us `mpc` allows for us to write normal code that just *looks like* a grammar, *or* we can use special notation to write a grammar directly!

Coding Grammars

So what does code that *looks like* a grammar..*look like*? Let us take a look at `mpc` by trying to write code for a grammar that recognizes the language of Shiba Inu. More colloquially know as *Doge*. This language we are going to define as follows.

› An *Adjective* is either *"wow"*, *"many"*, *"so"* or *"such"*. › A *Noun* is either *"lisp"*, *"language"*, *"c"*, *"book"* or *"build"*. › A *Phrase* is an *Adjective* followed by a *Noun*. › A *Doge* is zero or more *Phrases*.

We can start by trying to define *Adjective* and *Noun*. To do this we create two new parsers, represented by the type `mpc_parser_t*`, and we store them in the variables `Adjective` and `Noun`. We use the function `mpc_or` to create a parser where one of several options should be used, and the function `mpc_sym` to wrap our initial strings.

If you squint your eyes you could attempt to read the code as if it were the rules we specified above.

```
/* Build a new parser 'Adjective' to recognize descriptions */
mpc_parser_t* Adjective = mpc_or(4,
  mpc_sym("wow"), mpc_sym("many"),
  mpc_sym("so"),  mpc_sym("such")
);

/* Build a new parser 'Noun' to recognize things */
mpc_parser_t* Noun = mpc_or(5,
  mpc_sym("lisp"), mpc_sym("language"),
  mpc_sym("c"),    mpc_sym("book"),
  mpc_sym("build")
);
```

How can I access these `mpc` functions? For now don't worry about compiling or running any of the sample code in this chapter. Just read it and see if you can understand the theory behind grammars. In the next chapter we'll get setup with `mpc` and use it for a language closer to our Lisp.
To define `Phrase` we can reference our existing parsers. We need to use the function `mpc_and`, that specifies one thing is

required then another. As input we pass it `Adjective` and `Noun`, our previously defined parsers. This function also takes the arguments `mpcf_strfold` and `free`, which say how to join or delete the results of these parsers. Ignore these arguments for now.

```
mpc_parser_t* Phrase = mpc_and(2, mpcf_strfold, Adjective, Noun, free);
```

To define *Doge* we must specify that *zero or more* of some parser is required. For this we need to use the function `mpc_many`. As before, this function requires the special variable `mpcf_strfold` to say how the results are joined together, which we can ignore.

```
mpc_parser_t* Doge = mpc_many(mpcf_strfold, Phrase);
```

By creating a parser that looks for *zero or more* occurrences of another parser a cool thing has happened. Our `Doge` parser accepts inputs of any length. This means its language is *infinite*. Here are just some examples of possible strings `Doge` could accept. Just as we discovered in the first section of this chapter we have used a finite number of re-write rules to create an infinite language.

```
"wow book such language many lisp"
"so c such build such language"
"many build wow c"
""
"wow lisp wow c many language"
"so c"
```

If we use more `mpc` functions, we can slowly build up parsers that parse more and more complicated languages. The code we use *sort of* reads like a grammar, but becomes much more messy with added complexity. Due to this, taking this approach isn't always an easy task. A whole set of helper functions that build on simple constructs to make frequent tasks easy are all documented on the mpc repository. This is a good approach for complicated languages, as it allows for fine-grained control, but won't be required for our needs.

Natural Grammars

`mpc` lets us write grammars in a more natural form too. Rather than using C functions that look less like a grammar, we can specify the whole thing in one long string. When using this method we don't have to worry about how to join or discard inputs, with functions such as `mpcf_strfold`, or `free`. All of that is done automatically for us!

Here is how we would recreate the previous examples using this method.

```
mpc_parser_t* Adjective = mpc_new("adjective");
mpc_parser_t* Noun      = mpc_new("noun");
mpc_parser_t* Phrase    = mpc_new("phrase");
mpc_parser_t* Doge      = mpc_new("doge");

mpca_lang(MPC_LANG_DEFAULT,
  "                                          \
    adjective : \"wow\" | \"many\" | \"so\" | \"such\";            \
    noun      : \"lisp\" | \"language\" | \"c\" | \"book\" | \"build\"; \
    phrase    : <adjective> <noun>;                          \
    doge      : <phrase>*;                                \
  ",
  Adjective, Noun, Phrase, Doge);

/* Do some parsing here... */m

mpc_cleanup(4, Adjective, Noun, Phrase, Doge);
```

Without having an exactly understanding of the syntax for that long string, it should be obvious how much *clearer* the grammar is in this format. If we learn what all the special symbols mean we barely have to squint our eyes to read it as

one.

Another thing to notice is that the process is now in two steps. First we create and name several rules using `mpc_new` and then we define them using `mpca_lang`.

The first argument to `mpca_lang` are the options flags. For this we just use the defaults. The second is a long multi-line string in C. This is the *grammar* specification. It consists of a number of *re-write rules*. Each rule has the name of the rule on the left, a colon `:`, and on the right it's definition terminated with a semicolon `;`.

The special symbols used to define the rules on the right hand side work as follows.

`"ab"`	The string `ab` is required.
'a'	The character `a` is required.
'a' 'b'	First `'a'` is required, then `'b'` is required..
'a' | 'b'	Either `'a'` is required, or `'b'` is required.
'a'*	Zero or more `'a'` are required.
'a'+	One or more `'a'` are required.
`<abba>`	The rule called `abba` is required.

Sounds familiar... Did you notice that the description of what the input string to `mpca_lang` should look like sort of sounded like I was specifying a grammar? That's because it was! `mpc` uses itself internally to parse the input you give it to `mpca_lang`. It does it by specifying a *grammar* in code using the previous method. How neat is that..
Using what is described above verify that what I've written above is equal to what we specified in code.

This method of specifying a grammar is what we are going to use in the following chapters. It might seem overwhelming at first. Grammars can be difficult to understand right away. But as we continue you will become much more familiar with how to create and edit them.

This chapter is about theory, so if you are going to try some of the bonus marks, don't worry too much about correctness. Thinking in the right mindset is more important. Feel free to invent symbols and notation for certain concepts to make them simpler to write down. Some of the bonus marks also might require cyclic or recursive grammar stuctures, so don't worry if you have to use these!

Reference

doge_code.c
doge_grammar.c

Bonus Marks

- › Write down some more examples of strings the `Doge` language contains.
- › Why are there back slashes `\` in front of the quote marks `"` in the grammar?
- › Why are there back slashes `\` at the end of the line in the grammar?
- › Describe textually a grammar for decimal numbers such as `0.01` or `52.221`.
- › Describe textually a grammar for web URLs such as `http://www.buildyourownlisp.com`.
- › Describe textually a grammar for simple English sentences such as `the cat sat on the mat`.
- › Describe more formally the above grammars. Use `|`, `*`, or any symbols of your own invention.
- › If you are familiar with JSON, textually describe a grammar for it.

Parsing

Polish Notation

To try out `mpc` we're going to implement a simple grammar that resembles a mathematical subset of our Lisp. It's called Polish Notation and is a notation for arithmetic where the operator comes before the operands.

For example...

`1 + 2 + 6`	*is*	`+ 1 2 6`
`6 + (2 * 9)`	*is*	`+ 6 (* 2 9)`
`(10 * 2) / (4 + 2)`	*is*	`/ (* 10 2) (+ 4 2)`

We need to work out a grammar which describes this notation. We can begin by describing it *textually* and then later formalise our thoughts.

To start, we observe that in polish notation the operator always comes first in an expression, followed by either numbers or other expressions in parenthesis. This means we can say "a *program* is an *operator* followed by one or more *expressions*," where "an *expression* is either a *number*, or, in parenthesis, an *operator* followed by one or more *expressions*".

More formally...

`Program`	*the start of input*, an `Operator`, one or more `Expression`, and *the end of input*.
`Expression`	either a `Number` *or* `'('`, an `Operator`, one or more `Expression`, and an `')'`.
`Operator`	`'+'`, `'-'`, `'*'`, or `'/'`.
`Number`	an optional `-`, and one or more characters between `0` and `9`

Regular Expressions

We should be able to encode most of the above rules using things we know already, but *Number* and *Program* might

pose some trouble. They contain a couple of constructs we've not learnt how to express yet. We don't know how to express the start or the end of input, optional characters, or range of characters.

These can be expressed, but they require something called a *Regular Expression*. Regular expressions are a way of writing grammars for small sections of text such as words or numbers. Grammars written using regular expressions can't consist of multiple rules, but they do give precise, and concise, control over what is matched and what isn't. Here are some basic rules for writing regular expressions.

`a`	The character `a` is required.
`[abcdef]`	Any character in the set `abcdef` is required.
`[a-f]`	Any character in the range `a` to `f` is required.
`a?`	The character `a` is optional.
`a*`	Zero or more of the character `a` are required.
`a+`	One or more of the character `a` are required.
`^`	The start of input is required.
`$`	The end of input is required.

These are all the regular expression rules we need for now. Whole books have been written on learning regular expressions. For those curious much more information can be found online or from these sources. We will be using them in later chapters, so some basic knowledge will be required, but you won't need to master them for now.

In an `mpc` grammar we write regular expressions by putting them between forward slashes `/`. Using the above guide our *Number* rule can be expressed as a regular expression using the string `/-?[0-9]+/`.

Installing mpc

Before we work on writing this grammar we first need to *include* the `mpc` headers, and then *link* to the `mpc` library, just as we did for `editline` on Linux and Mac. Starting with your code from chapter 4, you can rename the file to `parsing.c` and download `mpc.h` and `mpc.c` from the mpc repo. Put these in the same directory as your source file.

To *include* `mpc` put `#include "mpc.h"` at the top of the file. To *link* to `mpc` put `mpc.c` directly into the compile command. On linux you will also have to link to the maths library by adding the flag `-lm`.

On **Linux** and **Mac**

```
cc -std=c99 -Wall parsing.c mpc.c -ledit -lm -o parsing
```

On **Windows**

```
cc -std=c99 -Wall parsing.c mpc.c -o parsing
```

Hold on, don't you mean `#include <mpc.h>`? There are actually two ways to include files in C. One is using angular brackets `<>` as we've seen so far, and the other is with quotation marks `""`. The only difference between the two is that using angular brackets searches the system locations for headers first, while quotation marks searches the current directory first. Because of this system headers such as `<stdio.h>` are typically put in angular brackets, while local headers such as `"mpc.h"` are typically put in quotation marks.

Polish Notation Grammar

Formalising the above rules further, and using some regular expressions, we can write a final grammar for the language

of polish notation as follows. Read the below code and verify that it matches what we had written textually, and our ideas of polish notation.

```
/* Create Some Parsers */
mpc_parser_t* Number   = mpc_new("number");
mpc_parser_t* Operator = mpc_new("operator");
mpc_parser_t* Expr     = mpc_new("expr");
mpc_parser_t* Lispy    = mpc_new("lispy");

/* Define them with the following Language */
mpca_lang(MPC_LANG_DEFAULT,
  "                                     \
    number   : /-?[0-9]+/ ;                    \
    operator : '+' | '-' | '*' | '/' ;         \
    expr     : <number> | '(' <operator> <expr>+ ')' ;  \
    lispy    : /^/ operator> expr>+ /$/ ;          \
  ",
  Number, Operator, Expr, Lispy);
```

We need to add this to the interactive prompt we started on in chapter 4. Put this code right at the beginning of the `main` function before we print the *Version* and *Exit* information. At the end of our program we also need to delete the parsers when we are done with them. Right before `main` returns we should place the following clean-up code.

```
/* Undefine and Delete our Parsers */
mpc_cleanup(4, Number, Operator, Expr, Lispy);
```

I'm getting an error `undefined reference to mpc_lang'` That should be `mpca_lang`, with an `a` at the end!

Parsing User Input

Our new code creates a `mpc` parser for our *Polish Notation* language, but we still need to actually *use* it on the user input supplied each time from the prompt. We need to edit our `while` loop so that rather than just echoing user input back, it actually attempts to parse the input using our parser. We can do this by replacing the call to `printf` with the following `mpc` code, that makes use of our program parser `Lispy`.

```
/* Attempt to Parse the user Input */
mpc_result_t r;
if (mpc_parse("stdin>";, input, Lispy, &;r)) {
  /* On Success Print the AST */
  mpc_ast_print(r.output);
  mpc_ast_delete(r.output);
} else {
  /* Otherwise Print the Error */
  mpc_err_print(r.error);
  mpc_err_delete(r.error);
}
```

This code calls the `mpc_parse` function with our parser `Lispy`, and the input string `input`. It copies the result of the parse into `r` and returns `1` on success and `0` on failure. We use the address of operator `&` on `r` when we pass it to the function. This operator will be explained in more detail in later chapters.

On success a internal structure is copied into `r`, in the field `output`. We can print out this structure using `mpc_ast_print` and delete it using `mpc_ast_delete`.

Otherwise there has been an error, which is copied into `r` in the field `error`. We can print it out using `mpc_err_print` and delete it using `mpc_err_delete`.

Compile these updates, and take this program for a spin. Try out different inputs and see how the system reacts. Correct behaviour should look like the following.

```
Lispy Version 0.0.0.0.2
Press Ctrl+c to Exit

lispy> + 5 (* 2 2)
>:
  regex:
  operator|char: '+'
  expr|number|regex: '5'
  expr|>:
    char: '('
    operator|char: '*'
    expr|number|regex: '2'
    expr|number|regex: '2'
    char: ')'
  regex:
lispy> hello
stdin>:0:0: error: expected '+', '-', '*' or '/' at 'h'
lispy> / 1dog & cat
stdin>:0:3: error: expected end of input at 'd'
lispy>
```

I'm getting an error `stdin>:0:0: error: Parser Undefined!`. This error is due to the syntax for your grammar supplied to `mpca_lang` being incorrect. See if you can work out what part of the grammar is incorrect. You can use the reference code for this chapter to help you find this, and verify how the grammar should look.

Reference

parsing.c

Bonus Marks

- › Write a regular expression matching strings of all `a` or `b` such as `aababa` or `bbaa`.
- › Write a regular expression matching strings of consecutive `a` and `b` such as `abababab` or `aba`.
- › Write a regular expression matching `pit`, `pot` and `respite` but *not* `peat`, `spit`, or `part`.
- › Change the grammar to add a new operator such as `%`.
- › Change the grammar to recognize operators written in textual format `add`, `sub`, `mul`, `div`.
- › Change the grammar to recognize decimal numbers such as `0.01`, `5.21`, or `10.2`.
- › Change the grammar to make the operators written conventionally, between two expressions.
- › Use the grammar from the previous chapter to parse `Doge`. You must add *start* and *end* of input!

Evaluation

Trees

Now we can read input, and we have it structured internally, but we are still unable to evaluate it. In this chapter we add the code that evaluates this structure and actually performs the computations encoded within.

This internal structure is what we saw printed out by the program in the previous chapter. It is called an *Abstract Syntax Tree*, and it represents the structure of the program based on the input entered by the user. At the leaves of this tree are numbers and operators - the actual data to be processed. At the branches are the rules used to produce this part of the tree - the information on how to traverse and evaluate it.

Before working out exactly how we are going to do this traversal, lets see exactly how this structure is defined internally. If we peek inside `mpc.h` we can have a look at the definition of `mpc_ast_t`, which is the data structure we got from the parse.

```
typedef struct mpc_ast_t {
  char* tag;
  char* contents;
  int children_num;
  struct mpc_ast_t** children;
} mpc_ast_t;
```

This struct has a number of fields we can access. Lets take a look at them one by one.

The first field is `tag`. When we printed out the tree this was the information that precluded the contents of the node. It was a string containing a list of all the rules used to parse that particular item. For example `expr|number|regex`.

This `tag` field is going to be important as it lets us see what type of thing the node is.

The second field is `contents`. This will contain the actual contents of the node such as `'*'`, `'('` or `'5'`. You'll notice for branches this is empty, but for leaves we can use it to find the operator or number to use.

Finally we see two fields that are going to help us traverse the tree. These are `children_num` and `children`. The first field tells us how many children a node has, and the second is an array of these children.

The type of the `children` field is `mpc_ast_t**`. This is a double pointer type. It isn't as scary as it looks and will be explained in greater detail in later chapters. For now you can think of it as a list of the child nodes of the this tree.

We can access a child node, by accessing this field using array notation. This is done by writing the field name `children` and suffixing it with square brackets containing the index of the child to access. For example to access the first child of the node we can use `children[0]`. Notice that C counts its array indices from `0`.

Because the type `mpc_ast_t*` is a *pointer* to a struct, there is a slightly different syntax to access its fields. We need to use an arrow `->` instead of a dot `.`. There is no fundamental reason for this switch in operators, so for now just remember that field access of pointer types uses an arrow.

```
/* Load AST from output */
mpc_ast_t* a = r.output;
printf("Tag: %s\n", a->tag);
printf("Contents: %s\n", a->contents);
printf("Number of children: %i\n", a->children_num);

/* Get First Child */
mpc_ast_t* c0 = a->children[0];
printf("First Child Tag: %s\n", c0->tag);
printf("First Child Contents: %s\n", c0->contents);
printf("First Child Number of children: %i\n", c0->children_num);
```

Recursion

There is a funny thing about this tree structure. It refers to itself. Each of its children are themselves trees again, and the children of those children are trees yet again. Just like our languages, and re-write rules, data in this structure contains repeated substructures, that resemble their parents.

This pattern of repeated substructures could go on and on. Clearly if we want a function which can work on all possible trees we can't look just a couple of nodes down, we have to define it to work on trees of any depth.

Luckily we can do this, by exploiting the nature of how these substructures repeat, and using a technique called *recursion*.

Put simply a *recursive function* is one that calls itself as some part of its calculation.

It sounds weird for a function to be defined in terms of itself. But consider that functions can give different outputs when supplied with different inputs. If we give some changed, or different inputs to a recursive call to the same function, and provide a way for this function to not call itself again under certain conditions, we can be more confident this *recursive function* is doing something useful.

As an example we can write a recursive function which will count the number of nodes in our tree structure.

To begin we work out how it will act in the most simple case, if the input tree has no children. In this case we know the result is simply one. Now we can go on to define the more complex case, if the tree has one or more children. In this case the result will be one (for the node itself), plus the number of nodes in all of those children.

But how do we find the number of nodes in all of the children? Well we can use the function we are in the process defining! *Yeah, Recursion.*

In C we might write it something like this.

```
int number_of_nodes(mpc_ast_t* t) {
 if (t->children_num == 0) { return 1; }
 if (t->children_num >= 1) {
  int total = 1;
  for (int i = 0; i < t->children_num; i++) {
   total = total + number_of_nodes(t->children[i]);
  }
  return total;
 }
}
```

Recursive functions are weird because they require an odd leap of faith. First we have to assume we have some function which does something correctly already, and then we have to go about using this function, to write the initial function we assumed we had!

Like most things, recursive functions almost always end up following a similar pattern. First a *base case* is defined. This is the case that ends the recursion, such as `t->children_num == 0` in our previous example. After this the *recursive case* is defined, such as `t->children_num >= 1` in our previous example, which breaks down a computation into smaller parts, and calls itself recursively to compute those parts, before combining them together.

Recursive functions can take some thought, so pause now and ensure you understand them before continuing onto other chapters because we'll be making good use of them in the rest of the book. If you are still uncertain, you can check out some of the bonus marks for this chapter to practice.

Evaluation

To evaluate the parse tree we are going to write a recursive function. But before we get started, let us try and see what observations we can make about the structure of the tree we get as input. Try printing out some expressions using your program from the previous chapter. See what you can notice.

```
lispy> + 1 (* 5 4)
>:
 regex:
 operator|char: '+'
 expr|number|regex: '1'
 expr|>:
  char: '('
  operator|char: '*'
  expr|number|regex: '5'
  expr|number|regex: '4'
  char: ')'
 regex:
```

One observation is that if a node is tagged with `number` it is always a number, has no children, and we can just convert the contents to an integer. This will act as the *base case* in our recursion.

If a node is tagged with `expr`, and is *not* a `number`, we need to look at its second child (the first child is always `'('`) and see which operator it is. Then we need to apply this operator to the *evaluation* of the remaining children, excluding the final child which is always `')'`. This is our *recursive case*. This also needs to be done for the root node.

When we evaluate our tree, just like when counting the nodes, we'll need to accumulate the result. To represent this result we'll use the C type `long` which means a *long integer*.

To detect the tag of a node, or to get a number from a node, we will need to make use of the `tag` and `contents` fields. These are *string* fields, so we are going to have to learn a couple of string functions first.

atoi	Converts a `char*` to a `long`.
strcmp	Takes as input two `char*` and if they are equal it returns `0`.
strstr	Takes as input two `char*` and returns a pointer to the location of the second in the first, or `0` if the second is not a sub-string of the first.

We can use `strcmp` to check which operator to use, and `strstr` to check if a tag contains some substring. Altogether our recursive evaluation function looks like this.

```
long eval(mpc_ast_t* t) {

 /* If tagged as number return it directly, otherwise expression. */
 if (strstr(t->tag, "number")) { return atoi(t->contents); }

 /* The operator is always second child. */
 char* op = t->children[1]->contents;

 /* We store the third child in `x` */
 long x = eval(t->children[2]);

 /* Iterate the remaining children, combining using our operator */
 int i = 3;
 while (strstr(t->children[i]->tag, "expr")) {
  x = eval_op(x, op, eval(t->children[i]));
  i++;
 }

 return x;
}
```

We can define the `eval_op` function as follows. It takes in a number, an operator string, and another number. It tests for which operator is passed in, an performs the corresponding C operation on the inputs.

```
/* Use operator string to see which operation to perform */
long eval_op(long x, char* op, long y) {
  if (strcmp(op, "+") == 0) { return x + y; }
  if (strcmp(op, "-") == 0) { return x - y; }
  if (strcmp(op, "*") == 0) { return x * y; }
  if (strcmp(op, "/") == 0) { return x / y; }
  return 0;
}
```

Printing

Instead of printing the tree we now want to print the result of the evaluation. Therefore we need to pass the tree into our `eval` function, and print the result we get using `printf` and the specifier `%li`, which is used for `long` type.

We also need to remember to delete the output tree after we are done evaluating it.

```
long result = eval(r.output);
printf("%li\n", result);
mpc_ast_delete(r.output);
```

If all of this is successful we should be able to do some basic maths with our new programming language!

```
Lispy Version 0.0.0.0.3
Press Ctrl+c to Exit

lispy> + 5 6
11
lispy> - (* 10 10) (+ 1 1 1)
97
lispy> - (/ 10 2) 20
-15
lispy>
```

Reference

evaluation.c

Bonus Marks

- › Write a recursive function to compute the number of leaves of a tree.
- › Write a recursive function to compute the number of branches of a tree.
- › Write a recursive function to compute the most number of children spanning from one branch of a tree.
- › How would you use `strstr` to see if a node was tagged as an `expr`.
- › How would you use `strcmp` to see if a node had the contents `'('` or `')'`.
- › Add the operator `%`, which returns the remainder of division. For example `% 10 6` is `4`.
- › Add the operator `^`, which raises one number to another. For example `^ 4 2` is `16`.
- › Add the function `min`, which returns the smallest number. For example `min 1 5 3` is `1`.
- › Add the function `max`, which returns the biggest number. For example `max 1 5 3` is `5`.
- › Change the minus operator `-` so that when it receives one argument it negates it.

Error Handling

Crashes

Some of you may have noticed a problem with the previous chapter's program. Try entering this into the prompt and see what happens.

```
Lispy Version 0.0.0.0.3
Press Ctrl+c to Exit

lispy> / 10 0
```

Ouch. The program crashed upon trying to divide by zero. It's okay if a program crashes during development, but our final program would hopefully never crash, and always explain to the user what went wrong.

At the moment our program can produce syntax errors but it still has no functionality for reporting errors in the evaluation of expressions. We need to build in some kind of error handling functionality to do this. It can be awkward in C, but if we start off on the right track, it will pay off later on when our system gets more complicated.

Lisp Value

There are several ways to deal with errors in C, but in this context my preferred method is to make errors a possible result of evaluating an expression. Then we can say that, in Lispy, an expression will evaluate to *either* a *number*, or an *error*. For example `+ 1 2` will evaluate to a number, but `/ 10 0` will evaluate to an error.

For this we need a data structure that can act as either one thing or anything. For simplicity sake we are just going to use a `struct` with fields specific to each thing that can be represented, and a special field `type` to tell us exactly what fields are meaningful to access.

This we are going to call an `lval`, which stands for *Lisp Value*.

```
/* Declare New lval Struct */
typedef struct {
  int type;
  long num;
  int err;
} lval;
```

Enumerations

You'll notice the type of the fields `type`, and `err`, is `int`. This means they are represented by a single integer number.

The reason we pick `int` is because we will assign meaning to each integer value, to encode what we require. For example we can make a rule *"If* `type` *is* `0` *then the structure is a Number."*, or *"If* `type` *is* `1` *then the structure is an Error."* This is a good way of doing things. It is simple and effective.

But if we litter our code with stray `0` and `1` then it is going to become increasingly unclear as to what is happening. Instead we can use named constants that have been assigned these integer values. This gives the reader an indication as to *why* one might be comparing a number to `0` or `1` and *what* is meant in this context.

In C this is supported using something called an `enum`.

```
/* Create Enumeration of Possible lval Types */
enum { LVAL_NUM, LVAL_ERR };
```

An `enum` is a declaration of variables which under the hood are automatically assigned integer constant values. Above is how we would declare some enumerated values for the `type` field.

We also want to declare an enumeration for the *error* field. We have three error cases in our particular program. There is division by zero, an unknown operator, or being passed a number that is too large to be represented internally using a `long`. These can be enumerated as follows.

```
/* Create Enumeration of Possible Error Types */
enum { LERR_DIV_ZERO, LERR_BAD_OP, LERR_BAD_NUM };
```

Lisp Value Functions

Our `lval` type is almost ready to go. Unlike the previous `long` type we have no current method for creating new instances of it. To do this we can declare two functions that construct an `lval` of either an *error* type or a *number* type.

```
/* Create a new number type lval */
lval lval_num(long x) {
  lval v;
  v.type = LVAL_NUM;
  v.num = x;
  return v;
}

/* Create a new error type lval */
lval lval_err(int x) {
  lval v;
  v.type = LVAL_ERR;
  v.err = x;
  return v;
}
```

These functions first create an `lval` called `v`, and assign the fields before returning it.

Because our `lval` function can now be one of two things we can no longer just use `printf` to output it. We will want to do different behaviour depending upon the type of the `lval` that is given. There is a concise way to do this in C using the `switch` statement. This takes some value as input and compares it to other known values, known as *cases*. When the values are equal it executes the code that follows up until the next `break` statement.

Using this we can build a function that can print an `lval` of any type like this.

```
/* Print an "lval" */
void lval_print(lval v) {
  switch (v.type) {
    /* In the case the type is a number print it, then 'break' out of the switch. */
    case LVAL_NUM: printf("%li", v.num); break;

    /* In the case the type is an error */
    case LVAL_ERR:
      /* Check What exact type of error it is and print it */
      if (v.err == LERR_DIV_ZERO) { printf("Error: Division By Zero!"); }
      if (v.err == LERR_BAD_OP)   { printf("Error: Invalid Operator!"); }
      if (v.err == LERR_BAD_NUM)  { printf("Error: Invalid Number!"); }
    break;
  }
}

/* Print an "lval" followed by a newline */
void lval_println(lval v) { lval_print(v); putchar('\n'); }
```

Evaluating Errors

Now that we know how to work with the `lval` type, we need to change our evaluation functions to use it instead of `long`.

As well as changing the type signatures we need to change the functions such that they work correctly upon encountering either an *error* as input, or a *number* as input.

In our `eval_op` function, if we encounter an error we should return it right away, and only do computation if both the arguments are numbers. We should modify our code to return an error rather than attempt to divide by zero. This will fix the crash from right at the beginning of this chapter.

```
lval eval_op(lval x, char* op, lval y) {

  /* If either value is an error return it */
  if (x.type == LVAL_ERR) { return x; }
  if (y.type == LVAL_ERR) { return y; }

  /* Otherwise do maths on the number values */
  if (strcmp(op, "+") == 0) { return lval_num(x.num + y.num); }
  if (strcmp(op, "-") == 0) { return lval_num(x.num - y.num); }
  if (strcmp(op, "*") == 0) { return lval_num(x.num * y.num); }
  if (strcmp(op, "/") == 0) {
    /* If second operand is zero return error instead of result */
    return y.num == 0 ? lval_err(LERR_DIV_ZERO) : lval_num(x.num / y.num);
  }

  return lval_err(LERR_BAD_OP);
}
```

What is that `?` doing there? You'll notice that for division to check if the second argument is zero we use a question mark symbol `?`, followed by a colon `:`. This is called the *ternary operator*, and it allows you to write conditional expressions on one line. It works something like this. `<condition> ? <then> : <else>`. In other words, if the condition is true it returns what follows the `?`, otherwise it returns what follows `:`. Some people dislike this operator because they believe it makes code unclear. If you are unfamiliar with the ternary operator, initially, you may feel awkward to use it; but once you get to know it there are rarely problems.
We need to give a similar treatment to our `eval` function. In this case because we've defined `eval_op` to robustly handle errors we just need to add the error conditions to our number conversion function.

In this case we use the `strtol` function to convert from string to `long`. This allows us to check a special variable `errno` to ensure the conversion goes correctly. This is a more robust way to convert numbers than our previous method using `atoi`.

```
lval eval(mpc_ast_t* t) {

  if (strstr(t->tag, "number")) {
    /* Check if there is some error in conversion */
    long x = strtol(t->contents, NULL, 10);
    return errno != ERANGE ? lval_num(x) : lval_err(LERR_BAD_NUM);
  }

  char* op = t->children[1]->contents;
  lval x = eval(t->children[2]);

  int i = 3;
  while (strstr(t->children[i]->tag, "expr")) {
    x = eval_op(x, op, eval(t->children[i]));
    i++;
  }

  return x;
}
```

The final small step is to change how we print the result found by our evaluation to use our newly defined printing function which can print any type of `lval`.

```
lval result = eval(r.output);
lval_println(result);
mpc_ast_delete(r.output);
```

And we are done! Try running this new program and make sure there are no crashes when dividing by zero!

```
lispy> / 10 0
Error: Division By Zero!
lispy> / 10 2
5
```

Plumbing

Some of you who have gotten this far in the book may feel uncomfortable with how it is progressing. You may feel you've managed to follow instructions well enough, but don't have a clear understanding of all of the underlying mechanisms going on behind the scenes.

If this is the case I want to reassure you that you are doing well. If you don't understand the internals it's because I have not explained everything in sufficient depth. This is okay.

To be able to progress and get code to work under these conditions is a really great skill in programming, and if you've made it this far it shows you have it.

In programming we call this *plumbing*. Roughly speaking this is following instructions to try and tie together a bunch libraries or components, without fully understanding how they work internally.

It requires *faith* and *intuition*. *Faith* is required to believe that if the stars align, and every incantation is correctly performed for this magical machine, the right thing will really happen. And *intuition* is required to work out what has gone wrong, and how to fix things when they don't go as planned.

Unfortunately these can't be taught directly, so if you've made it this far then *congratulations!* You've made it over a difficult hump, and in the following chapters I promise we'll finish up with the plumbing, and actually start to programming that feels fresh and wholesome.

Reference

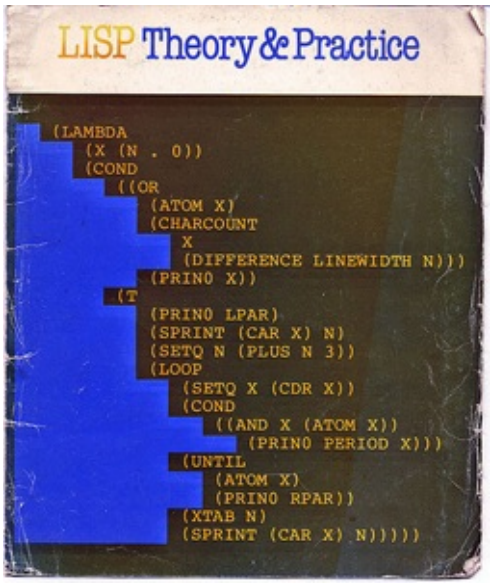
error_handling.c

Bonus Marks

- › How do you give an `enum` a name?
- › What are `union` data types and how do they work?
- › Can you change how `lval` is defined to use `union` instead of `struct`?
- › What are the advantages over using a `union` instead of `struct`?
- › Extend parsing and evaluation to support the remainder operator `%`.
- › Extend parsing and evaluation to support decimal types using a `double` field.

S-Expressions

Lists and Lisps



Lisps are famous for having little distinction between data and code. They use the same structures to represent both. This allows them to do many powerful things which other languages cannot do. If we want this power for our programming language we're going to have to separate out the process of *reading* in input, and *evaluating* that input we have stored.

The final result of this chapter will only differ slightly in behaviour from the previous chapter. This is because we are going to spend time changing how things work internally. This is called *re-factoring* and it will make our life a lot easier later on. Like preparation for a meal, just because we're not putting food onto plates it doesn't mean we're wasting time. Sometimes the anticipation is even better than eating!

To store the program we will need to create an internal list structure that is built up recursively of numbers, symbols, and other lists. In Lisp, this structure is commonly called an S-Expression standing for *Symbolic Expression*. We will extend our `lval` structure to be able to represent it. The evaluation behaviour of S-Expressions is the behaviour typical of Lisps, that we are used to so far. To evaluate an S-Expression we look at the first item in the list, and take this to be the operator. We then look at all the other items in the list, and take these as operands to get the result.

By introducing S-Expressions we'll finally be entering the world of Lisp.

Pointers

In C no concept of lists can be explored without dealing properly with pointers. Pointers are a famously misunderstood aspect of C. They are difficult to teach because while being conceptually very simple, they come with a lot of new terminology, and often no clear use-case. This makes them appear far more monstrous than they are. Luckily for us, we have a couple ideal use-cases, both of which are extremely typical in C, and will likely end up being how you use pointers 90% of the time.

The reason we need pointers in C is because of how function calling works. When you call a function in C the arguments are always passed *by value*. This means *a copy* of them is passed to the function you call. This is true for `int`, `long`, `char`, and user defined `struct` types such as `lval`. Most of the time this is great but occasionally it can cause issues.

A common problem is if we have a large struct containing many other sub structs we wish to pass around. Every time we call a function we must create another copy of it. Suddenly the amount of data that needs to be copied around just to call a function can become huge!

A second problem is this. When we define a `struct`, it is always a fixed size. It has a limited number of fields, and each of

these fields must be a struct which itself is limited in size. If I want to call a function with just *a list of things*, where the number of *things* varies from call to call, clearly I can't use a `struct` to do this.

To get around these issues the developers of C (or y'know...someone) came up with a clever idea. They imagined computer memory as a single huge list of bytes. In this list each byte can be given a global index, or position. A bit like a house number. The first byte is numbered `0`, the second is `1`, etc.

In this case, all the data in the computer, including the structs and variables used in the currently running program, start at some index in this huge list. If, rather than copying the data itself to a function, we instead copy a number representing the *index* at where this data starts, the function being called can look up any amount of data it wants.

By using *addresses* instead of the actual data, we can allow a function to access and modify some location in memory without having to copy any data. Functions can also use pointers to do other cool stuff, like output data to some address given as input.

Because the total size of computer memory is fixed, the number of bytes needed to represent an address is always the same. But if we keep track of it, the number of bytes the address points to can grow and shrink. This means we can create a variable sized data-structure and still sort of *pass* it to a function, which can inspect and modify it.

So a pointer is just a number. A number representing the starting index of some data in memory. The type of the pointer hints to us, and the compiler, what type of data might be accessible at this location.

We can declare pointer types by suffixing existing ones with the the `*` character. We've seen some examples of this already with `mpc_parser_t*`, `mpc_ast_t*`, or `char*`.

To create a pointer to some data, we need to get its index, or *address*. To get the address of a some data we use the *address of* operator `&`. Again you've seen this before when we passed in a pointer to `mpc_parse` so it would output into our `mpc_result_t`.

Finally to get the data at an address, called *dereferencing*, we use the `*` operator on the left hand size of a variable. To get the data at the field of a pointer to a struct we use the arrow `->`. This you saw in chapter 7.

The Stack & The Heap

I said that memory can be visualized of as one long list of bytes. Actually it is better to imagine it split into two sections. These sections are called *The Stack* and *The Heap*.

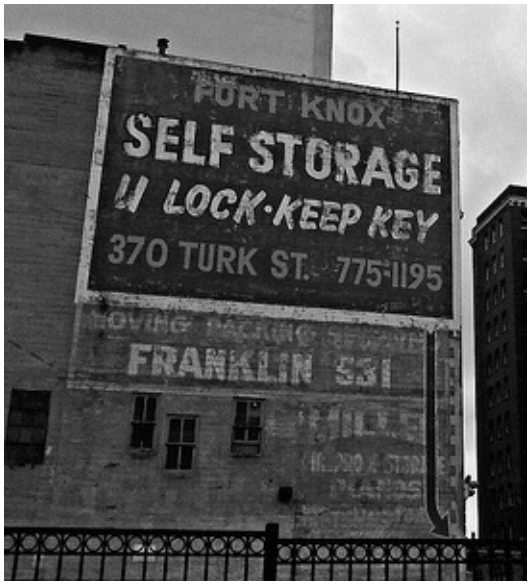
Some of you may have heard tales of these mysterious locations, such as *"the stack grows down but the heap grows up"*, or *"there can be many stacks, but only one heap"*. These sorts of things don't matter much. Dealing with the stack and the heap in C can be complex, but it doesn't have to be a mystery. In essence they are just two sections of memory used for two different tasks.

The Stack

The Stack is the memory where your program lives. It is where all of your temporary variables and data structures exist as you manipulate and edit them. Each time you call a function a new area of the stack is put aside for it to use. Into this area are put local variables, copies of any arguments passed to the function, as well as some bookkeeping data such as who the caller was, and what to do when finished. When the function is done the area it used is unallocated, ready for use again by someone else.

I like to think of the stack as a building site. Each time we need to do something new we corner off a section of space, enough for our tools and materials, and set to work. We can still go to other parts of the site, or go off-site, if we need certain things, but all our work is done in this section. Once we are done with some task, we take what we've constructed to a new place and clean up that section of the space we've been using to make it.

The Heap



The Heap is a section of memory put aside for storage of objects with a longer lifespan. Memory in this area has to be manually allocated and deallocated. To allocate new memory the `malloc` function is used. This function takes as input the number of bytes required, and returns back a pointer to a new block of memory with that many bytes set aside.

When done with the memory at that location it must be released again. To do this the pointer received from `malloc` should be passed to the `free` function.

Using the Heap is trickier than the Stack because it requires the programmer to remember to call `free` and to call it correctly. If he or she doesn't, the program may continuously allocate more and more memory. This is called a *memory*

leak. A easy rule to avoid this is to ensure for each `malloc` there is a corresponding (and only one corresponding) `free`. If this can always be ensured the program should be handling The Heap correctly.

I Imagine the Heap like a huge U-Store-It. We can call up the reception with `malloc` and request a number of boxes. With these boxes we can do what we want, and we know they will persist no matter how messy the building site gets. We can take things to and from the U-Store-It and the building site. It is useful to store materials and large objects which we only need to retrieve once in a while. The only problem is we need to remember to call the receptionist again with `free` when we are done. Otherwise soon we'll have requested all the boxes, have no space, and run up a huge bill.

Parsing Expressions

Because we're now thinking in S-Expressions, and not Polish Notation we need to update our parser. The syntax for S-Expressions is simple. It is just a number of other Expressions between parenthesis, where an Expression can be a Number, Operator, or other S-Expression . We can modify our existing parse rules to reflect this. We also are going to rename our `operator` rule to `symbol`. This is in anticipation of adding more operators, variables and functions later.

```
mpc_parser_t* Number = mpc_new("number");
mpc_parser_t* Symbol = mpc_new("symbol");
mpc_parser_t* Sexpr  = mpc_new("sexpr");
mpc_parser_t* Expr   = mpc_new("expr");
mpc_parser_t* Lispy  = mpc_new("lispy");

mpca_lang(MPC_LANG_DEFAULT,
  "                                     \
    number : /-?[0-9]+/ ;               \
    symbol : '+' | '-' | '*' | '/' ;    \
    sexpr  : '(' <expr>* ')' ;          \
    expr   : <number> | <symbol> | <sexpr> ; \
    lispy  : /^/ <expr>* /$/ ;          \
  ",
  Number, Symbol, Sexpr, Expr, Lispy);
```

We should also remember to cleanup these rules on exit.

```
mpc_cleanup(5, Number, Symbol, Sexpr, Expr, Lispy);
```

Expression Structure

We need a way to store S-Expressions internally as `lval`. This means we'll also need to store internally *Symbols* and *Numbers*. We're going to add two new `lval` types to the enumeration. The first new type is `LVAL_SYM`, which we're going to use to represent operators such as `+`. The second new type is `LVAL_SEXPR` which we're going to use to represent S-Expressions.

```
/* Add SYM and SEXPR as possible lval types. */
enum { LVAL_ERR, LVAL_NUM, LVAL_SYM, LVAL_SEXPR };
```

S-Expressions are variable length *lists* of other S-Expressions. As we learnt at the beginning of this chapter we can't create variable length structs, so we are going to need to use a pointer. We are going to create a pointer field `cell` which points to a location where we store a list of `lval*`. These are pointers to the other individual `lval`. Our field should therefore be a double pointer type `lval**`. A *pointer to* `lval` *pointers*.

We will also need to keep track of how many `lval*` are in this list, so we add an extra field `count` to record this.

Are there ever pointers to pointers to pointers? There is an old programming joke (and by *programming joke* I mean *silly anecdote*), which says you can rate C programmers by how many stars are on their pointers. Beginners program might only use `char*` or the odd `int*`, so they were called *one star programmer*. Most intermediate programs contain

double pointer types such as `lval**`. These programmers are therefore called *two star programmers*. To spot a triple pointer is something special. You would be viewing the work of someone grand and terrible, writing code not meant to be read with mortal eyes. As such being called a *three star programmer* is rarely a compliment. As far as I know, a quadruple pointer has never been seen in the wild.
To represent symbols we're going to use a string. We're also going to change the representation of errors to a string. This means we can store a unique error message rather than just an error code. This will make our error reporting better and more flexible, and we can get rid of the original error `enum`. Our updated `lval` struct looks like this.

```
typedef struct lval {
  int type;

  long num;

  /* Error and Symbol types have some string data */
  char* err;
  char* sym;

  /* Count and Pointer to a list of "lval*" */
  int count;
  struct lval** cell;

} lval;
```

What is that `struct` keyword doing there? Our new definition of `lval` needs to contain a reference to itself. This means we have to slightly change how it is defined. Before we open the curly brackets we can put the name of the struct, and then refer to this inside using `struct lval`. Even though a struct can refer to its own type, it must only contain pointers to its own type, not its own type directly. Otherwise the size of the struct would refer to itself, and grow infinite in size when you tried to calculate it!

Constructors & Destructors

We can change our `lval` construction functions to return pointers to an `lval`, rather than one directly. This will make keeping track of `lval` variables easier. For this we need to use `malloc` with the `sizeof` function to allocate enough space for the `lval` struct, and then to fill in the fields with the relevant information using the arrow operator `->`.

When we construct an `lval` its fields may contain pointers to other things that have been allocated on the heap. This means we need to be careful. Whenever we are done with an `lval` we also need to delete the things it points to on the heap. We will have to make a rule to ourselves. Whenever we free the memory allocated for an `lval`, we also free all the things it points to.

```
/* Construct a pointer to a new Number lval */
lval* lval_num(long x) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_NUM;
  v->num = x;
  return v;
}

/* Construct a pointer to a new Error lval */
lval* lval_err(char* m) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_ERR;
  v->err = malloc(strlen(m) + 1);
  strcpy(v->err, m);
  return v;
}

/* Construct a pointer to a new Symbol lval */
lval* lval_sym(char* s) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_SYM;
  v->sym = malloc(strlen(s) + 1);
  strcpy(v->sym, s);
  return v;
}

/* A pointer to a new empty Sexpr lval */
lval* lval_sexpr(void) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_SEXPR;
  v->count = 0;
  v->cell = NULL;
  return v;
}
```

What is `NULL`? `NULL` is a special constant that points to memory location `0`. In many places it is used as a convention to signify some non-value or empty data. Above we use it to specify that we have a data pointer, but that it doesn't point to any number of data items. You will see `NULL` crop up a lot more later on so always remember to look out for it.
Why are you using `strlen(s) + 1`? In C strings are *null terminated*. This means that the final character of them is always the zero character `\0`. This is a convention in C to signal the end of a string. It is important that all strings are stored this way otherwise programs will break in nasty ways. The `strlen` function only returns the number of bytes in a string *excluding* the null terminator. This is why we need to add one, to ensure there is enough allocated space for it all!
We now need a special function to delete `lval*`. This should call `free` on the pointer itself to release the memory acquired from `malloc`, but more importantly it should inspect the type of the `lval`, and release any memory pointed to by its fields. If we match every call to one of the above construction functions with `lval_del` we can ensure we will get no memory leaks.

```
void lval_del(lval* v) {

  switch (v->type) {
    /* Do nothing special for number type */
    case LVAL_NUM: break;

    /* For Err or Sym free the string data */
    case LVAL_ERR: free(v->err); break;
    case LVAL_SYM: free(v->sym); break;

    /* If Sexpr then delete all elements inside */
    case LVAL_SEXPR:
      for (int i = 0; i < v->count; i++) {
        lval_del(v->cell[i]);
      }
      /* Also free the memory allocated to contain the pointers */
      free(v->cell);
    break;
  }

  /* Finally free the memory allocated for the "lval" struct itself */
  free(v);
}
```

Reading Expressions

First we are going to *read* in the program and construct an `lval*` that represents it all. Then we are going to *evaluate* this `lval*` to get the result of our program. This first stage should convert the *abstract syntax tree* into an S-Expression, and the second stage should evaluate this S-Expression using our normal Lisp rules.

To complete the first stage we can recursively look at each node of the tree, and construct different `lval*` types depending on the `tag` and `contents` fields of the node.

If the given node is tagged as a `number` or `symbol`, then we use our constructors to return an `lval*` directly for those types. If the given node is the `root`, or an `sexpr`, then we create an empty S-Expression `lval` and slowly add each valid sub-expression contained in the tree.

To add an element to an S-Expression we can create a function `lval_add`. This function increases the count of the Expression list by one, and then uses `realloc` to reallocate the amount of space required by `v->cell`. This new space can be used to store the extra `lval*` required. Using this new space it sets the final value of the list with `v->cell[v->count-1]` to the value `lval* x` passed in.

Don't Lisps use Cons cells? Other Lisps have a slightly different definition of what an S-Expression is. In most other Lisps S-Expressions are defined inductively as either an *atom* such as a symbol of number, or two other S-Expressions joined, or *cons*, together. This naturally leads to an implementation using *linked lists*, a different data structure to the one we are using. I choose to represent S-Expressions as a variable sized array in this book for the purposes of simplicity, but it is important to be aware that the official definition, and typical implementation are both subtly different.

```
lval* lval_add(lval* v, lval* x) {
  v->count++;
  v->cell = realloc(v->cell, sizeof(lval*) * v->count);
  v->cell[v->count-1] = x;
  return v;
}

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

lval* lval_read(mpc_ast_t* t) {

  /* If Symbol or Number return conversion to that type */
  if (strstr(t->tag, "number")) { return lval_read_num(t); }
  if (strstr(t->tag, "symbol")) { return lval_sym(t->contents); }

  /* If root (>) or sexpr then create empty list */
  lval* x = NULL;
  if (strcmp(t->tag, ">") == 0) { x = lval_sexpr(); }
  if (strstr(t->tag, "sexpr"))  { x = lval_sexpr(); }

  /* Fill this list with any valid expression contained within */
  for (int i = 0; i < t->children_num; i++) {
    if (strcmp(t->children[i]->contents, "(") == 0) { continue; }
    if (strcmp(t->children[i]->contents, ")") == 0) { continue; }
    if (strcmp(t->children[i]->contents, "}") == 0) { continue; }
    if (strcmp(t->children[i]->contents, "{") == 0) { continue; }
    if (strcmp(t->children[i]->tag,  "regex") == 0) { continue; }
    x = lval_add(x, lval_read(t->children[i]));
  }

  return x;
}
```

Printing Expressions

We are so close to trying out all of our new changes! We need to modify our print function to print out S-Expressions types. Using this we can double check that the *reading* phase is working correctly by printing out the S-Expressions we read in and verifying they match those we input.

To print out S-Expressions we can create another function that loops over all the sub-expressions of an expression and prints these individually separated by spaces, just like how they are input.

```
void lval_expr_print(lval* v, char open, char close) {
 putchar(open);
 for (int i = 0; i < v->count; i++) {

  /* Print Value contained within */
  lval_print(v->cell[i]);

  /* Don't print trailing space if last element */
  if (i != (v->count-1)) {
   putchar(' ');
  }
 }
 putchar(close);
}

void lval_print(lval* v) {
 switch (v->type) {
  case LVAL_NUM:   printf("%li", v->num); break;
  case LVAL_ERR:   printf("Error: %s", v->err); break;
  case LVAL_SYM:   printf("%s", v->sym); break;
  case LVAL_SEXPR: lval_expr_print(v, '(', ')'); break;
 }
}

void lval_println(lval* v) { lval_print(v); putchar('\n'); }
```

I can't declare these functions because they call each other. The `lval_expr_print` function calls the `lval_print` function and vice-versa. There is no way we can order them in the source file to resolve this dependency. Instead we need to *forward declare* one of them. This is declaring a function without giving it a body. It lets other functions call it, while allowing you to define it properly later on. To write a forward declaration write the function definition but instead of the body put a semicolon `;`. In this example we should put `void lval_print(lval* v);` somewhere in the source file before `lval_expr_print`. You'll definitely run into this later, and I won't always alert you about it, so try to remember how to fix it! In our main loop, we can remove the evaluation for now, and instead try reading in the result and printing out what we have read.

```
lval* x = lval_read(r.output);
lval_println(x);
lval_del(x);
```

If this is successful you should see something like the following when entering input to your program.

```
lispy> + 2 2
(+ 2 2)
lispy> + 2 (* 7 6) (* 2 5)
(+ 2 (* 7 6) (* 2 5))
lispy> *    55    101  (+ 0 0 0)
(* 55 101 (+ 0 0 0))
lispy>
```

Evaluating Expressions

The behaviour of our evaluation function is largely the same as before. We need to adapt it to deal with `lval*` and our more relaxed definition of what constitutes an expression. We can think of our evaluation function as a kind of transformer. It takes in some `lval*` and transforms it in some way to some new `lval*`. In some cases it can just return exactly the same thing. In other cases it may modify the input `lval*` and the return it. In many cases it will delete the input, and return something completely different. If we are going to return something new we must always remember to delete the `lval*` we get as input.

For S-Expressions we first evaluate all the children of the S-Expression. If any of these children are errors we return the

first error we encounter using a function we'll define later called `lval_take`.

If the S-Expression has no children we just return it directly. This corresponds to the empty expression, denoted by `()`. We also check for single expressions. These are expressions with only one child such as `(5)`. In this case we return the single expression contained within the parenthesis.

If neither of these are the case we know we have a valid expression with more than one child. In this case we separate the first element of the expression using a function we'll define later called `lval_pop`. We then check this is a *symbol* and not anything else. If it is a symbol we check what symbol it is, and pass it, and the arguments, to a function `builtin_op` which does our calculations. If the first element is not a symbol we delete it, and the values passed into the evaluation function, returning an error.

To evaluate all other types we just return them directly back.

```
lval* lval_eval_sexpr(lval* v) {

  /* Evaluate Children */
  for (int i = 0; i < v->count; i++) {
    v->cell[i] = lval_eval(v->cell[i]);
  }

  /* Error Checking */
  for (int i = 0; i < v->count; i++) {
    if (v->cell[i]->type == LVAL_ERR) { return lval_take(v, i); }
  }

  /* Empty Expression */
  if (v->count == 0) { return v; }

  /* Single Expression */
  if (v->count == 1) { return lval_take(v, 0); }

  /* Ensure First Element is Symbol */
  lval* f = lval_pop(v, 0);
  if (f->type != LVAL_SYM) {
    lval_del(f); lval_del(v);
    return lval_err("S-expression Does not start with symbol!");
  }

  /* Call builtin with operator */
  lval* result = builtin_op(v, f->sym);
  lval_del(f);
  return result;
}

lval* lval_eval(lval* v) {
  /* Evaluate Sexpressions */
  if (v->type == LVAL_SEXPR) { return lval_eval_sexpr(v); }
  /* All other lval types remain the same */
  return v;
}
```

There are two functions we've used and not defined in the above code. These are `lval_pop` and `lval_take`. These are general purpose functions for manipulating S-Expression `lval` types which we'll make use of here and in the future.

The `lval_pop` function extracts a single element from an S-Expression at index `i` and shifts the rest of the list backward so that it no longer contains that `lval*`. It then returns the extracted value. Notice that it doesn't delete the input list. It is like taking an element from a list and popping it out, leaving what remains. This means both the element popped and the old list need to be deleted at some point with `lval_del`.

The `lval_take` function is similar to `lval_pop` but it deletes the list it has extracted the element from. This is like taking an element from the list and deleting the rest. It is a slight variation on `lval_pop` but it makes our code easier to read in some places. Unlike `lval_pop`, only the expression you take from the list needs to be deleted by `lval_del`.

```
lval* lval_pop(lval* v, int i) {
  /* Find the item at "i" */
  lval* x = v->cell[i];

  /* Shift the memory following the item at "i" over the top of it */
  memmove(&v->cell[i], &v->cell[i+1], sizeof(lval*) * (v->count-i-1));

  /* Decrease the count of items in the list */
  v->count--;

  /* Reallocate the memory used */
  v->cell = realloc(v->cell, sizeof(lval*) * v->count);
  return x;
}

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

We also need to define the evaluation function `builtin_op`. This is like the `eval_op` function used in our previous chapter but modified to take a single `lval*` representing a list of all the arguments to operate on. It needs to do some more rigorous error checking. If any of the inputs are a non-number `lval*` we need to return an error.

First it checks that all the arguments input are numbers. It then pops the first argument to start. If there are no more sub-expressions and the operator is subtraction it performs unary negation on this first number. This makes expressions such as `(- 5)` evaluate correctly.

If there are more arguments it constantly pops the next one from the list and performs arithmetic depending on which operator we're meant to be using. If a zero is encountered on division it deletes the temporary `x`, `y`, and the argument list `a`, and returns an error.

If there have been no errors the input arguments are deleted and the new expression returned.

```
lval* builtin_op(lval* a, char* op) {

  /* Ensure all arguments are numbers */
  for (int i = 0; i < a->count; i++) {
    if (a->cell[i]->type != LVAL_NUM) {
      lval_del(a);
      return lval_err("Cannot operator on non number!");
    }
  }

  /* Pop the first element */
  lval* x = lval_pop(a, 0);

  /* If no arguments and sub then perform unary negation */
  if ((strcmp(op, "-") == 0) && a->count == 0) { x->num = -x->num; }

  /* While there are still elements remaining */
  while (a->count > 0) {

    /* Pop the next element */
    lval* y = lval_pop(a, 0);

    /* Perform operation */
    if (strcmp(op, "+") == 0) { x->num += y->num; }
    if (strcmp(op, "-") == 0) { x->num -= y->num; }
    if (strcmp(op, "*") == 0) { x->num *= y->num; }
    if (strcmp(op, "/") == 0) {
      if (y->num == 0) {
        lval_del(x); lval_del(y); lval_del(a);
        x = lval_err("Division By Zero!"); break;
      } else {
        x->num /= y->num;
      }
    }

    /* Delete element now finished with */
    lval_del(y);
  }

  /* Delete input expression and return result */
  lval_del(a);
  return x;
}
```

This completes our evaluation functions. We just need to change `main` again so it passes the input through this evaluation before printing it.

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

Now you should now be able to evaluate expressions correctly in the same way as in the previous chapter. Take a little breather and have a play around with the new evaluation. Make sure everything is working correctly, and the behaviour is as expected. In the next chapter we're going to make great use of these changes to implement some cool new features.

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

Reference

s_expressions.c

Bonus Marks

- › Give an example of a variable in our program that lives on *The Stack*.
- › Give an example of a variable in our program that points to *The Heap*.
- › What does the `strcpy` function do?
- › What does the `realloc` function do?
- › What does the `memmove` function do?
- › How does `memmove` differ from `memcpy`?
- › Extend parsing and evaluation to support the remainder operator `%`.
- › Extend parsing and evaluation to support decimal types using a `double` field.

Q-Expressions

Adding Features

You'll notice that the following chapters will all follow a similar pattern. This pattern is the typical approach used to add new features to a language. It consists of a number of steps that bring a feature from start to finish. These are listed below, and are exactly what we're going to do in this chapter, to introduce a new feature called a Q-Expression.

Step 0 • Syntax	Add new rule to the language grammar for this feature.
Step 1 • Representation	Add new data type variation to represent this feature.
Step 2 • Parsing	Add new functions for reading this feature from the *abstract syntax tree*.
Step 3 • Semantics	Add new functions for evaluating and manipulating this feature.

Quoted Expressions

In this chapter we'll implement a new type of Lisp Value called a Q-Expression.

This stands for *quoted expression*, and is a type of Lisp Expression that is not evaluated by the standard Lisp mechanics. When encountered by the evaluation function Q-expressions are left exactly as they are. This makes them ideal for a number of purposes. We can use them to store and manipulate other Lisp values such as numbers, symbols, or other S-Expressions themselves!

After we've added Q-Expressions we are going to implement a concise set of operators to manipulate them. Like the arithmetic operators these will prove fundamental in how we think about and play with expressions.

The syntax for Q-Expressions is very similar to that of S-Expressions. The only difference is that instead of parenthesis `()` Q-Expressions are surrounded by curly brackets `{}`. We can add this to our grammar as follows.

I've never heard of Q-Expressions. Q-Expressions don't exist in other Lisps. Other Lisps use *Macros* to stop evaluation. These look like normal functions, but they do not evaluate their arguments. A special Macro called quote `'` exists, which can be used to stop the evaluation of almost anything. This is the inspiration for Q-Expressions, which are unique to our Lisp, and will be used instead of Macros for doing all the same tasks and more. The way I've used *S-Expression* and *Q-Expression* in this book is a slight abuse of terminology, but I hope this misdemeanor makes the behaviour of our Lisp clearer.

```
mpc_parser_t* Number = mpc_new("number");
mpc_parser_t* Symbol = mpc_new("symbol");
mpc_parser_t* Sexpr  = mpc_new("sexpr");
mpc_parser_t* Qexpr  = mpc_new("qexpr");
mpc_parser_t* Expr   = mpc_new("expr");
mpc_parser_t* Lispy  = mpc_new("lispy");

mpca_lang(MPC_LANG_DEFAULT,
  "                                     \
    number : /-?[0-9]+/ ;                    \
    symbol : '+' | '-' | '*' | '/' ;         \
    sexpr  : '(' <expr>* ')' ;               \
    qexpr  : '{' <expr>* '}' ;               \
    expr   : <number> | <symbol> | <sexpr> | <qexpr> ; \
    lispy  : /^/ <expr>* /$/ ;               \
  ",
  Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
```

We also must remember to update our cleanup function to deal with the new rule we've added.

```
mpc_cleanup(6, Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
```

Reading Q-Expressions

Because Q-Expressions are so similar S-Expressions much of their internal behaviour is going to be the same. We're going to reuse our S-Expression data fields to represent Q-Expressions, but we still need to add a separate type to the enumeration.

```
enum { LVAL_ERR, LVAL_NUM, LVAL_SYM, LVAL_SEXPR, LVAL_QEXPR };
```

We can also add a constructor for this variation.

```
/* A pointer to a new empty Qexpr lval */
lval* lval_qexpr(void) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_QEXPR;
  v->count = 0;
  v->cell = NULL;
  return v;
}
```

To print and delete Q-Expressions we do essentially the same thing as with S-Expressions. We can add the relevant lines to our functions for printing and deletion as follows.

```
void lval_del(lval* v) {

  switch (v->type) {
    case LVAL_NUM: break;
    case LVAL_ERR: free(v->err); break;
    case LVAL_SYM: free(v->sym); break;

    /* If Qexpr or Sexpr then delete all elements inside */
    case LVAL_QEXPR:
    case LVAL_SEXPR:
      for (int i = 0; i < v->count; i++) {
        lval_del(v->cell[i]);
      }
      /* Also free the memory allocated to contain the pointers */
      free(v->cell);
    break;
  }

  free(v);
}
```

```
void lval_print(lval* v) {
  switch (v->type) {
    case LVAL_NUM:   printf("%li", v->num); break;
    case LVAL_ERR:   printf("Error: %s", v->err); break;
    case LVAL_SYM:   printf("%s", v->sym); break;
    case LVAL_SEXPR: lval_expr_print(v, '(', ')'); break;
    case LVAL_QEXPR: lval_expr_print(v, '{', '}'); break;
  }
}
```

Using these simple changes we can update our reading function `lval_read` to be able to read in in Q-Expressions. Because we reused all the S-Expression data fields for our Q-Expression type, we can also reuse all of the functions for S-Expressions such as `lval_add`. Therefore to read in Q-Expressions we just need to add a special case for constructing an empty Q-Expression to `lval_read` just below where we detect and create empty S-Expressions from the *abstract syntax tree*.

```
if (strstr(t->tag, "qexpr"))  { x = lval_qexpr(); }
```

Because there is no special method of evaluating Q-Expressions, we don't need to edit any of the evaluation functions. Our Q-Expressions should be ready to try. Compile and run the program. See if you can use them as a new data type. Ensure they are not evaluated.

```
lispy> {1 2 3 4}
{1 2 3 4}
lispy> {1 2 (+ 5 6) 4}
{1 2 (+ 5 6) 4}
lispy> {{2 3 4} {1}}
{{2 3 4} {1}}
lispy>
```

Builtin Functions

We can read in Q-Expressions but they are still somewhat useless. We need some way to manipulate them.

For this we can define some built-in operators to work on our list type. Choosing a concise set of these is important. If we implement a few fundamental operations then we can use these to define new operations without adding extra C code. There are a few ways to pick these fundamental operators but I've chosen a set that will allow us to do everything we need. They are defined as follows.

`list`	Takes one or more arguments and returns a new Q-Expression containing the arguments
`head`	Takes a Q-Expression and returns a Q-Expression with only of the first element
`tail`	Takes a Q-Expression and returns a Q-Expression with the first element removed
`join`	Takes one or more Q-Expressions and returns a Q-Expression of them conjoined together
`eval`	Takes a Q-Expression and evaluates it as if it were a S-Expression

Like with our mathematical operators we should add these functions as possible valid symbols. Afterward we can go about trying to define their behaviour in a similar way to `builtin_op`.

```
mpca_lang(MPC_LANG_DEFAULT,
 "                                                    \
  number : /-?[0-9]+/ ;                               \
  symbol : \"list\" | \"head\" | \"tail\" | \"join\" | \"eval\" | '+' | '-' | '*' | '/' ; \
  sexpr  : '(' <expr>* ')' ;                          \
  qexpr  : '{' <expr>* '}' ;                          \
  expr   : <number> | <symbol> | <sexpr> | <qexpr> ;                \
  lispy  : /^/ <expr>* /$/ ;                          \
 ",
 Number, Symbol, Sexpr, Qexpr, Expr, Lispy)
```

First Attempt

Our builtin functions should have the same interface as `builtin_op`. That means the arguments should be bundled into an S-Expression which the function must use and then delete. They should return a new `lval*` as a result of the evaluation.

The actual functionality of taking the head or tail of an Q-Expression shouldn't be too hard for us. We can make use of the existing functions we've defined for S-Expressions such as `lval_take` and `lval_pop`. But like `builtin_op` we also need to check that the inputs we get are valid.

Lets take a look at `head` and `tail` first. These functions have a number of conditions under which they can't act. First of all we must ensure they are only passed a single argument, and that that argument is a Q-Expression. Then we need to ensure that this Q-Expression isn't empty and actually has some elements.

The `head` function can repeatedly pop and delete the item at index `1` until there is nothing else left in the list.

The `tail` function is even more simple. It can pop and delete the item at index `0`, leaving the tail remaining. An initial attempt at these functions might look like this.

```
lval* builtin_head(lval* a) {
  /* Check Error Conditions */
  if (a->count != 1) {
    lval_del(a);
    return lval_err("Function 'head' passed too many arguments!");
  }

  if (a->cell[0]->type != LVAL_QEXPR) {
    lval_del(a);
    return lval_err("Function 'head' passed incorrect types!");
  }

  if (a->cell[0]->count == 0) {
    lval_del(a);
    return lval_err("Function 'head' passed {}!");
  }

  /* Otherwise take first argument */
  lval* v = lval_take(a, 0);

  /* Delete all elements that are not head and return */
  while (v->count > 1) { lval_del(lval_pop(v, 1)); }
  return v;
}

lval* builtin_tail(lval* a) {
  /* Check Error Conditions */
  if (a->count != 1) {
    lval_del(a);
    return lval_err("Function 'tail' passed too many arguments!");
  }

  if (a->cell[0]->type != LVAL_QEXPR) {
    lval_del(a);
    return lval_err("Function 'tail' passed incorrect types!");
  }

  if (a->cell[0]->count == 0) {
    lval_del(a);
    return lval_err("Function 'tail' passed {}!");
  }

  /* Take first argument */
  lval* v = lval_take(a, 0);

  /* Delete first element and return */
  lval_del(lval_pop(v, 0));
  return v;
}
```

Macros

These `head` and `tail` functions do the correct thing, but the code pretty unclear, and long. There is so much error checking that the functionality is hard to see. One method we can use to clean it up is to use a *Macro*.

A Macro is a *preprocessor* statement for creating function-like-things that are evaluated before the program is compiled. It can be used for many different things, one of which is what we need to do here, clean up code.

Macros work by taking some arguments (which can be almost anything), and copying and pasting them into some given pattern. By changing the pattern or the arguments we can change what different code is generated by the Macro. To define macros we use the `#define` preprocessor directive. After this we write the name of the macro, followed by the argument names in parenthesis. After this the pattern is specified, to declare what code should be generated for the given arguments.

We can design a macro to help with our error conditions called `LASSERT`. Macros are typically given names in all caps to help distinguish them from normal C functions. This macro take in three arguments `args`, `cond` and `err`. It then generates code as shown on the right hand side, but with these variables pasted in at the locations where they are named. This pattern is a good fit for all of our error conditions.

```
#define LASSERT(args, cond, err) if (!(cond)) { lval_del(args); return lval_err(err); }
```

We can use this to change how our above functions are written, without actually changing what code is generated by the compiler. This makes it much easier to read for the programmer, and saves a bit of typing. The rest of the error conditions for our functions should become a piece of cake to write too!

Head & Tail

Using this our `head` and `tail` functions are defined as follows. Notice how much clearer their real functionality is.

```
lval* builtin_head(lval* a) {
  LASSERT(a, (a->count == 1             ), "Function 'head' passed too many arguments!");
  LASSERT(a, (a->cell[0]->type == LVAL_QEXPR), "Function 'head' passed incorrect type!");
  LASSERT(a, (a->cell[0]->count != 0      ), "Function 'head' passed {}!");

  lval* v = lval_take(a, 0);
  while (v->count > 1) { lval_del(lval_pop(v, 1)); }
  return v;
}

lval* builtin_tail(lval* a) {
  LASSERT(a, (a->count == 1             ), "Function 'tail' passed too many arguments!");
  LASSERT(a, (a->cell[0]->type == LVAL_QEXPR), "Function 'tail' passed incorrect type!");
  LASSERT(a, (a->cell[0]->count != 0      ), "Function 'tail' passed {}!");

  lval* v = lval_take(a, 0);
  lval_del(lval_pop(v, 0));
  return v;
}
```

List & Eval

The `list` function is dead simple. It just converts the input S-Expression to a Q-Expression and returns it.

The `eval` function is somewhat like the opposite. It takes as input some single Q-Expression, which it converts to an S-Expression, and evaluates using `lval_eval`.

```
lval* builtin_list(lval* a) {
  a->type = LVAL_QEXPR;
  return a;
}

lval* builtin_eval(lval* a) {
  LASSERT(a, (a->count == 1              ), "Function 'eval' passed too many arguments!");
  LASSERT(a, (a->cell[0]->type == LVAL_QEXPR), "Function 'eval' passed incorrect type!");

  lval* x = lval_take(a, 0);
  x->type = LVAL_SEXPR;
  return lval_eval(x);
}
```

Join

The `join` function is our final function to define.

Unlike the others it can take multiple arguments, so its structure looks somewhat more like that of `builtin_op`. First we check that all of the arguments are Q-Expressions and then we join them together one-by-one. To do this we use the function `lval_join`. This works by repeatedly popping each item from `y` and adding it to `x` until `y` is empty. It then deletes `y` and returns `x`.

```
lval* builtin_join(lval* a) {

  for (int i = 0; i < a->count; i++) {
    LASSERT(a, (a->cell[i]->type == LVAL_QEXPR), "Function 'join' passed incorrect type.");
  }

  lval* x = lval_pop(a, 0);

  while (a->count) {
    x = lval_join(x, lval_pop(a, 0));
  }

  lval_del(a);
  return x;
}

lval* lval_join(lval* x, lval* y) {

  /* For each cell in 'y' add it to 'x' */
  while (y->count) {
    x = lval_add(x, lval_pop(y, 0));
  }

  /* Delete the empty 'y' and return 'x' */
  lval_del(y);
  return x;
}
```

Builtins Lookup

We've now got all of our builtin functions defined. We need to make a function that can call the correct one depending on what symbol it encounters in evaluation. We can do this using `strcmp` and `strstr`.

```
lval* builtin(lval* a, char* func) {
  if (strcmp("list", func) == 0) { return builtin_list(a); }
  if (strcmp("head", func) == 0) { return builtin_head(a); }
  if (strcmp("tail", func) == 0) { return builtin_tail(a); }
  if (strcmp("join", func) == 0) { return builtin_join(a); }
  if (strcmp("eval", func) == 0) { return builtin_eval(a); }
  if (strstr("+-/*", func)) { return builtin_op(a, func); }
  lval_del(a);
  return lval_err("Unknown Function!");
}
```

Then we can change our evaluation line in `lval_eval_sexpr` to call `builtin` rather than `builtin_op`.

```
/* Call builtin with operator */
lval* result = builtin(v, func->sym);
lval_del(func);
return result;
```

Finally Q-Expressions should be fully supported in our language! Compile and run the latest version and see what you can do with the new list operators. Try putting code and symbols into our lists and evaluating them in different ways. The ability to put S-Expressions inside a list using Q-Expressions is pretty awesome. It means we can treat code like data itself. This is a flagship feature of Lisps, and something that really cannot be done in languages such as C!

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

Reference

q_expressions.c

Bonus Marks

- › What are the four typical steps for adding new language features?
- › Create a Macro specifically for testing for the incorrect number of arguments.
- › Create a Macro specifically for testing for being called with the empty list.
- › Add a builtin function `cons` that takes a value and a Q-Expression and appends it to the front.
- › Add a builtin function `len` that returns the number of elements in a Q-Expression.
- › Add a builtin function `init` that returns all of a Q-Expression except the final element.

Variables

Immutability

In the previous chapters we've learnt a lot, and made great progress on the infrastructure of our language.

Already we can do a number of cool things that other languages can't, such as putting code inside of lists. Now is the time to start adding in the *features* which will make our language practical. The first one of these is going to be *variables*.

They're called variables, but it's a misleading name, because our variables won't vary. Our variables are *immutable* meaning they cannot change. Everything in our language so far has acted like it is *immutable*. When we evaluate an expression we have imagined that the old thing has been deleted and a new thing returned. In implementation often it is easier for us to reuse the data from the old thing to build the new thing, but conceptually it is a good way to think about how our language works.

So actually our variables are simply a way of *naming values*. They let us assign a *name* to a *value*, and then let us get a copy of that *value* later on when we need it.

To allow for *naming values* we need to create a structure which stores the name and value of everything named in our program. We call this the *environment*. When we start a new interactive prompt we want to create a new environment to go along with it, in which each new bit of input is evaluated. Then we can store and recall variables as we program.

What happens when we re-assign a name to something new? Isn't this like mutability? In our Lisp, when we re-assign a name we're going to delete the old association and create a new one. This gives the illusion that the thing assigned to that name has changed, and is mutable, but in fact we have deleted the old thing and assigned it a new thing. This is different to C where we really can change the data pointed to by a pointer, or stored in a struct, without deleting it and creating a new one.

Symbol Syntax

Now that we're going to allow for user defined variables we need to update the grammar for symbols to be more flexible. Rather than just our builtin functions it should match any possible valid symbol. Unlike in C, where the name a variable can be given is fairly restrictive, we're going to allow for all sorts of characters in the name of a variable.

We can create a regular expression that expresses the range of characters available as follows.

```
/[a-zA-Z0-9_+\\-*\\/\\\\=<>!&]+/
```

On first glace this looks like we've just bashed our hands in the keyboard. Actually it is a regular expression using a big

range specifier `[]`. Inside range specifiers special characters lose their meaning, but some of these characters still need to be escaped with backslashes. Because this is part of a C string we need to put two back slashes to represent a single backslash character in the input.

This rule lets symbols be any of, the standard C identifier characters `a-zA-Z0-9_`, the arithmetic operator characters `+\\-*\\/`, the backslash character `\\\\`, the comparison operator characters `=<>!`, or an ampersands `&`. This will give us all the flexibility we need for defining new and existing symbols.

```
mpca_lang(MPC_LANG_DEFAULT,
 "                                    \
  number : /-?[0-9]+/ ;                \
  symbol : /[a-zA-Z0-9_+\\-*\\/\\\\=<>!&]+/ ;  \
  sexpr  : '(' <expr>* ')' ;           \
  qexpr  : '{' <expr>* '}' ;           \
  expr   : <number> | <symbol> | <sexpr> | <qexpr> ; \
  lispy  : /^/ <expr>* /$/ ;           \
 ",
 Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
```

Function Pointers

Once we introduce variables, symbols will no longer represent functions in our language, but rather they will represent a name for us to lookup into our environment and get some new value back from.

Therefore we need a new value to represent functions in our language, which we can return once one of the builtin symbols is encountered. To create this new type of `lval` we are going to use something called a *function pointer*.

Function pointers are a great feature of C that lets you store and pass around pointers to functions. It doesn't make sense to edit the data pointed to by these pointers. Instead we use them to call the function they point to, as if it were a normal function.

Like normal pointers, function pointers have some type associated with them. This type specifies the type of the function pointed to, not the type of the data pointed to. This lets the compiler work out if it has been called correctly.

In the previous chapter our builtin functions took a `lval*` as input and returned a `lval*` as output. In this chapter our builtin functions will take an extra pointer to the environment `lenv*` as input. We can declare a new function pointer type called `lbuiltin`, for this type of function, like this.

```
typedef lval*(*lbuiltin)(lenv*, lval*);
```

Why is that syntax so odd? In some places the syntax of C can look particularly weird. It can help if we understand exactly why the syntax is like this. Let us de-construct the above syntax part by part. First the `typedef`. This can be put before any standard variable declaration. It results in the name of the variable, being declared a new type, matching what would be the inferred type of that variable. This is why in the above declaration what looks like the function name becomes the new type name. Next all those `*`. Pointer types in C are actually meant to be written with the star `*` on the left hand side of the variable name, not the right hand side of the type `int *x;`. This is because C type syntax works by a kind of weird inference. Instead of reading *"Create a new* `int` *pointer* `x`*"*. It is meant to read *"Create a new variable* `x` *where to dereference* `x` *results in an* `int`*."* Therefore `x` is inferred to be a pointer to an `int`. This idea is extended to function pointers. We can read the above declaration as follows. "To get an `lval*` we dereference `lbuiltin` and call it with a `lenv*` and a `lval*`." Therefore `lbuiltin` must be a function pointer that takes an `lenv*` and a `lval*` and returns a `lval*`.

Cyclic Types

The `lbuiltin` type references the `lval` type and the `lenv` type. This means that they should be declared first in the source file.

But we want to make a `lbuiltin` field in our `lval` struct so we can create function values. So therefore our `lbuiltin` declaration must go before our `lval` declaration. This leads to what is called a cyclic type dependency, where two types depend on each other.

We've come across this problem before with functions which depend on each other. The solution was to create a *forward declaration* which declared a function but left the body of it empty.

In C we can do exactly the same with types. First we declare two `struct` types without a body. Secondly we typedef these to the names `lval` and `lenv`. Then we can define our `lbuiltin` function pointer type. And finally we can define the body of our `lval` struct. Now all our type issues are resolved and the compiler won't complain any more.

```
/* Forward Declarations */

struct lval;
struct lenv;
typedef struct lval lval;
typedef struct lenv lenv;

/* Lisp Value */

enum { LVAL_ERR, LVAL_NUM, LVAL_SYM, LVAL_FUN, LVAL_SEXPR, LVAL_QEXPR };

typedef lval*(*lbuiltin)(lenv*, lval*);

struct lval {
  int type;

  long num;
  char* err;
  char* sym;
  lbuiltin fun;

  int count;
  lval** cell;
};
```

Function Type

As we've added a new possible `lval` type with the enumeration `LVAL_FUN`. We should update all our relevant functions that work on `lvals` to deal correctly with this update. In most cases this just means inserting new cases into switch statements.

We can start by making a new constructor function for this type.

```
lval* lval_fun(lbuiltin func) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_FUN;
  v->fun = func;
  return v;
}
```

On **deletion** we don't need to do anything special for function pointers.

```
case LVAL_FUN: break;
```

On **printing** we can just print out some nominal string.

```
case LVAL_FUN:  printf("<function>"); break;
```

We're also going to add a new function for **copying** an `lval`. This is going to come in useful when we put things into, and

take things out of, the environment. For numbers and functions we can just copy the relevant fields directly. For strings we need to copy using `malloc` and `strcpy`. To copy lists we need to allocate the correct amount of space and then copy each element individually.

```
lval* lval_copy(lval* v) {

  lval* x = malloc(sizeof(lval));
  x->type = v->type;

  switch (v->type) {

    /* Copy Functions and Numbers Directly */
    case LVAL_FUN: x->fun = v->fun; break;
    case LVAL_NUM: x->num = v->num; break;

    /* Copy Strings using malloc and strcpy */
    case LVAL_ERR: x->err = malloc(strlen(v->err) + 1); strcpy(x->err, v->err); break;
    case LVAL_SYM: x->sym = malloc(strlen(v->sym) + 1); strcpy(x->sym, v->sym); break;

    /* Copy Lists by copying each sub-expression */
    case LVAL_SEXPR:
    case LVAL_QEXPR:
      x->count = v->count;
      x->cell = malloc(sizeof(lval*) * x->count);
      for (int i = 0; i < x->count; i++) {
        x->cell[i] = lval_copy(v->cell[i]);
      }
    break;
  }

  return x;
}
```

Environment

Our environment structure must encode a list of relationships between *names* and *values*. There are many ways to build a structure that can do this sort of thing. We are going to go for the simplest possible method that works well. This is to use two lists of equal length. One is a list of `lval*`, and the other is a list of `char*`. Each entry in one list has a corresponding entry in the other list at the same position.

We've already forward declared our `lenv` struct, so we can define it as follows.

```
struct lenv {
  int count;
  char** syms;
  lval** vals;
};
```

We need some functions to create and delete this structure. These are pretty simple. Creation initialises the struct fields, while deletion iterates over the items in both lists and deletes or frees them.

```
lenv* lenv_new(void) {
  lenv* e = malloc(sizeof(lenv));
  e->count = 0;
  e->syms = NULL;
  e->vals = NULL;
  return e;
}

void lenv_del(lenv* e) {
  for (int i = 0; i < e->count; i++) {
    free(e->syms[i]);
    lval_del(e->vals[i]);
  }
  free(e->syms);
  free(e->vals);
  free(e);
}
```

Next we can create two functions that either get values from the environment or put values into it.

To get a value from the environment we loop over all the items in the environment and check if the given symbol matches any of the stored strings. If we find a match we can return a copy of the stored value. If no match is found we should return an error.

The function for putting new variables into the environment is a little bit more complex. First we want to check if a variable with the same name already exists. If this is the case we should replace its value with the new one. To do this we loop over all the existing variables in the environment and check their name. If a match is found we delete the value stored at that location, and store there a copy of the input value.

If no existing value is found with that name, we need to allocate some more space to put it in. For this we can use `realloc`, and store a copy of the `lval` and its name at the newly allocated locations.

```
lval* lenv_get(lenv* e, lval* k) {

  /* Iterate over all items in environment */
  for (int i = 0; i < e->count; i++) {
    /* Check if the stored string matches the symbol string */
    /* If it does, return a copy of the value */
    if (strcmp(e->syms[i], k->sym) == 0) { return lval_copy(e->vals[i]); }
  }
  /* If no symbol found return error */
  return lval_err("unbound symbol!",);
}

void lenv_put(lenv* e, lval* k, lval* v) {

  /* Iterate over all items in environment */
  /* This is to see if variable already exists */
  for (int i = 0; i < e->count; i++) {

    /* If variable is found delete item at that position */
    /* And replace with variable supplied by user */
    if (strcmp(e->syms[i], k->sym) == 0) {
      lval_del(e->vals[i]);
      e->vals[i] = lval_copy(v);
      e->syms[i] = realloc(e->syms[i], strlen(k->sym)+1);
      strcpy(e->syms[i], k->sym);
      return;
    }
  }

  /* If no existing entry found then allocate space for new entry */
  e->count++;
  e->vals = realloc(e->vals, sizeof(lval*) * e->count);
  e->syms = realloc(e->syms, sizeof(char*) * e->count);

  /* Copy contents of lval and symbol string into new location */
  e->vals[e->count-1] = lval_copy(v);
  e->syms[e->count-1] = malloc(strlen(k->sym)+1);
  strcpy(e->syms[e->count-1], k->sym);
}
```

Variable Evaluation

Our evaluation function now depends on the some environment. We should pass this in as an argument and use it to extract get a value if we encounter a symbol type.

```
lval* lval_eval(lenv* e, lval* v) {
  if (v->type == LVAL_SYM)   { return lenv_get(e, v); }
  if (v->type == LVAL_SEXPR) { return lval_eval_sexpr(e, v); }
  return v;
}
```

Because we've added a function type, our evaluation of S-Expressions also needs to change. Instead of checking for a symbol type we want to ensure it is a function type. If this condition holds we can call the `fun` field of the `lval` using the same notation as standard function calls.

```
lval* lval_eval_sexpr(lenv* e, lval* v) {

  for (int i = 0; i < v->count; i++) { v->cell[i] = lval_eval(e, v->cell[i]); }
  for (int i = 0; i < v->count; i++) { if (v->cell[i]->type == LVAL_ERR) { return lval_take(v, i); } }

  if (v->count == 0) { return v; }
  if (v->count == 1) { return lval_take(v, 0); }

  /* Ensure first element is a function after evaluation */
  lval* f = lval_pop(v, 0);
  if (f->type != LVAL_FUN) {
    lval_del(v); lval_del(f);
    return lval_err("first element is not a function");
  }

  /* If so call function to get result */
  lval* result = f->fun(e, v);
  lval_del(f);
  return result;
}
```

Builtins

Now that our evaluation relies on the new function type we need to make sure we can register all of our builtin functions with the environment before we start the interactive prompt. At the moment our builtin functions might not be the correct type. We need to change their type signature such that they take in some environment, and where appropriate change them to pass this environment into other calls that require it.

Once we've changed them to the correct type we can create a function that registers all of our builtins into an environment.

For each builtin we want to create a function `lval` and symbol `lval` with the given name. We then register these with the environment using `lenv_put`. The environment always takes or returns copies of a values, so we need to remember to delete these two `lval` after registration as we won't need them anymore.

If we split this task into two functions we can neatly register all of our builtins with some environment.

```
void lenv_add_builtin(lenv* e, char* name, lbuiltin func) {
  lval* k = lval_sym(name);
  lval* v = lval_fun(func);
  lenv_put(e, k, v);
  lval_del(k); lval_del(v);
}

void lenv_add_builtins(lenv* e) {
  /* List Functions */
  lenv_add_builtin(e, "list", builtin_list);
  lenv_add_builtin(e, "head", builtin_head); lenv_add_builtin(e, "tail",  builtin_tail);
  lenv_add_builtin(e, "eval", builtin_eval); lenv_add_builtin(e, "join",  builtin_join);

  /* Mathematical Functions */
  lenv_add_builtin(e, "+",    builtin_add); lenv_add_builtin(e, "-",     builtin_sub);
  lenv_add_builtin(e, "*",    builtin_mul); lenv_add_builtin(e, "/",     builtin_div);
}
```

The final step is to call this function before we create the interactive prompt. We also need to remember to delete the environment once we are finished.

```
lenv* e = lenv_new();
lenv_add_builtins(e);

while (1) {

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

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

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

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

  free(input);

}

lenv_del(e);
```

If everything is working correctly we should have a play around in the prompt and verify that functions are actually a new type of value now, not symbols.

```
lispy> +
<function>
lispy> eval (head {5 10 11 15})
5
lispy> eval (head {+ - + - * /})
<function>
lispy> (eval (head {+ - + - * /})) 10 20
30
lispy> hello
Error: unbound symbol!
lispy>
```

Define Function

We've managed to register our builtins as variables but we still don't have a way for users to define their own variables.

This is actually a bit awkward. We need to get the user to pass in a symbol to name, as well as the value to assign to it.

But symbols can't appear on their own. Otherwise the evaluation function will attempt to retrieve a value for them from the environment.

The only way we can pass around symbols without them being evaluated is to put them between `{}` in a quoted expression. So we're going to use this technique for our define function. It will take as input a list of symbols, and a number of other values. It will then assign each of the values to each of the symbols.

This function should act like any other builtin. It first checks for error conditions and then performs some command and return a value. In this case it first checks that the input arguments are the correct types. It then iterates over each symbol and value and puts them into the environment. If these is an error we can return this, but on success we will return the empty expression `()`.

```
lval* builtin_def(lenv* e, lval* a) {
  LASSERT(a, (a->cell[0]->type == LVAL_QEXPR), "Function 'def' passed incorrect type!");

  /* First argument is symbol list */
  lval* syms = a->cell[0];

  /* Ensure all elements of first list are symbols */
  for (int i = 0; i < syms->count; i++) {
    LASSERT(a, (syms->cell[i]->type == LVAL_SYM), "Function 'def' cannot define non-symbol");
  }

  /* Check correct number of symbols and values */
  LASSERT(a, (syms->count == a->count-1), "Function 'def' cannot define incorrect number of values to symbols");

  /* Assign copies of values to symbols */
  for (int i = 0; i < syms->count; i++) {
    lenv_put(e, syms->cell[i], a->cell[i+1]);
  }

  lval_del(a);
  return lval_sexpr();
}
```

We need to register this new builtin using our builtin function `lenv_add_builtins`.

```
/* Variable Functions */
lenv_add_builtin(e, "def",  builtin_def);
```

Now we should be able to support user defined variables! Because our `def` function takes in a list of symbols we can do some cool things storing and manipulating symbols in lists before passing them to be defined. Have a play around in the prompt and ensure everything is working correctly. You should get behaviour as follows. See what other complex methods are possible for the definition and evaluation of variables. Once we get to defining functions we'll really see some of the neat things that can be done with this approach.

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

Error Reporting

So far our error reporting kind of sucks. We can report when an error occurs, and give a vague notion of what was the problem was, but we don't give the user much information about what exactly has gone wrong. For example if there is an unbound symbol we should be able to report exactly which symbol was unbound. This can help the user track down errors, typos, and other trivial problems.

Wouldn't it be great if we could write a function that can report errors in a similar way to how `printf` works. It would be ideal if we could pass in strings, integers, and other data to make our error messages richer.

The `printf` function is a special function in C because it takes a variable number of arguments. We can create our own *variable argument* functions, which is what we're going to do to make our error reporting better.

We'll modify `lval_err` to act in the same way as `printf`, taking in a format string, and after that a variable number of arguments to match into this string.

To declare that a function takes variables arguments in the type signature you use the special syntax of ellipsis `...`, which represent the rest of the arguments.

```
lval* lval_err(char* fmt, ...);
```

Then, inside the function there are some standard library functions we can use to examine what the caller has passed in.

The first step is to create a `va_list` struct and initialise it with `va_start`, passing in the last named argument. For other

purposes it is possible to examine each argument passed in using `va_arg`, but we are going to pass our whole variable argument list directly to the `vsnprintf` function. This function acts like `printf` but instead writes to a string and takes in a `va_list`. Once we are done with our variable arguments, we shoulder call `va_end` to cleanup any resources used.

The `vsnprintf` function outputs to a string, which we need to allocate some first. Because we don't know the size of this string until we've run the function we first allocate a buffer `512` characters big and then reallocate to a smaller buffer once we've output to it. If an error message is going to be longer than 512 character it will just get cut off, but hopefully this won't happen.

Putting it all together our new error function looks like this.

```
lval* lval_err(char* fmt, ...) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_ERR;

  /* Create a va list and initialize it */
  va_list va;
  va_start(va, fmt);

  /* Allocate 512 bytes of space */
  v->err = malloc(512);

  /* printf into the error string with a maximum of 511 characters */
  vsnprintf(v->err, 511, fmt, va);

  /* Reallocate to number of bytes actually used */
  v->err = realloc(v->err, strlen(v->err)+1);

  /* Cleanup our va list */
  va_end(va);

  return v;
}
```

Using this we can then start adding in some better error messages to our functions. As an example we can look at `lenv_get`. When a symbol can't be found, rather than reporting a generic error, we can actually report the name that was not found.

```
return lval_err("Unbound Symbol '%s'", k->sym);
```

We can also adapt our `LASSERT` macro such that it can take variable arguments too. Because this is a macro and not a standard function the syntax is slightly different. On the left hand side of the definition we use the ellipses notation again, but on the right hand side we use a special variable `__VA_ARGS__` to paste in the contents of all the other arguments.

We need to prefix this special variable with two hash signs `##`. This ensure that it is pasted correctly when the macro is passed no extra arguments. In essence what this does is make sure to remove the leading comma `,` to appear as if no extra arguments were passed in.

Because we might use `args` in the construction of the error message we need to make sure we don't delete it until we've created the error value.

```
#define LASSERT(args, cond, fmt, ...) \
  if (!(cond)) { lval* err = lval_err(fmt, ##__VA_ARGS__); lval_del(args); return err; }
```

Now we can update some of our error messages to make them more informative. For example if the incorrect number of arguments were passed we can specify how many were required and how many were given.

```
LASSERT(a, (a->count == 1), "Function 'head' passed too many arguments. Got %i, Expected %i.", a->count, 1);
```

We can also improve our error reporting for type errors. We should attempt to report what type was expected by a function

and what type it actually got. Before we can do this it would be useful to have a function that took as input some type enumeration and returned a string representation of that type.

```
char* ltype_name(int t) {
 switch(t) {
   case LVAL_FUN: return "Function";
   case LVAL_NUM: return "Number";
   case LVAL_ERR: return "Error";
   case LVAL_SYM: return "Symbol";
   case LVAL_SEXPR: return "S-Expression";
   case LVAL_QEXPR: return "Q-Expression";
   default: return "Unknown";
 }
}
```

Then we can use this function in our `LASSERT` functions to report what was retrieved and what was expected in a useful way.

```
LASSERT(a, (a->cell[0]->type == LVAL_QEXPR),
  "Function 'head' passed incorrect type for argument 0. Got %s, Expected %s.",
  ltype_name(a->cell[0]->type), type_name(LVAL_QEXPR));
```

Go ahead and improve the error reporting in all situations where we return an error in the code. This should make debugging many of the next stages much easier as we begin to write real, and complicated code using our new language! See if you can use macros to save on typing and automatically generate code for common methods of error reporting.

```
lispy> + 1 {5 6 7}
Error: Function '+' passed incorrect type for argument 1. Got Q-Expression, Expected Number.
lispy> head {1 2 3} {4 5 6}
Error: Function 'head' passed incorrect number of arguments. Got 2, Expected 1.
lispy>
```

Reference

variables.c

Bonus Marks

- › Create a Macro to aid specifically with reporting type errors.
- › Create a Macro to aid specifically with reporting argument count errors.
- › Create a Macro to aid specifically with reporting empty list errors.
- › Change printing a builtin function print its name.
- › Write a function for printing out all the named values in an environment.
- › Redefine one of the builtin variables to something different.
- › Change redefinition of one of the builtin variables to something different an error.
- › Create an `exit` function for stopping the prompt and exiting.

Functions

What is a Function?

Functions are the essence of all programming. Back in the early days of computer science they represented a naive dream. The idea was that we could reduce computation into these smaller and smaller bits of re-usable code. Given enough time, and a proper structure for libraries, eventually we would have written code required for all computational needs. No longer would people have to write their own functions, and programming would consist of an easy job of stitching together components.

This dream hasn't come true yet, but it persists, no matter how flawed. Each new programming technique, or paradigm, that comes along shakes up this idea a little. They promise better re-use of code. Better abstractions, and an easier life for all.

In reality what each paradigm delivers is simply *different* abstractions. There has always been a trade-off. For each higher level of thinking about programming, some piece is thrown away. And this means, no matter how well you decide what to keep and what to leave, occasionally someone will need that piece that has been lost. But through all of this, one way or the other, functions have always persisted, and continually proven to be effective.

We've used functions in C, we know what they *look like*, but we don't know exactly what they *are*. Here are a few ways to think about them.

One way to think about functions is as description of some computation you want to be performed later. When you define a function it is like saying "when I use *this* name I want *that* sort of thing to happen". This is a very practical idea of a function. It is very intuitive, and metaphorical to language. This is the way you would command a human, or animal. Another thing I like about this is that it captures the delayed nature of functions. Functions are defined once, but can be called on repeatedly after.

Another way to think about functions is as a black box that takes some input and produces some output. This idea is subtly different from the former. It is more algebraic, and doesn't talk about *computation* or *commands*. This idea is a mathematical concept, and is not tied to some particular machine, or language. In some situations this idea is exceptionally useful. It allows us to think about functions without worrying about their internals, or how they are computed exactly. We can then combine and compose them together without worry of something subtle going wrong. This is the core idea behind an abstraction, and is what allows layers of complexity to work together with each other rather than conflict. This idea's strength can also be its downfall. Because it does not mention anything about computation it does not deal with a number of real world concerns. *"How long will this function take to run?"*, *"Is this function efficient?"*, *"Will it modify the state of my program? If so how?"*.

A third method is to think of functions as *partial computations*. Like the Mathematical model they can take some inputs. These values are required before it can complete the computation. This is why it is called *partial*. But like the computational model, the body of the function consists of a computation specified in some language of commands. These inputs are called *unbound variables*, and to finish the computation one simply supplies them. Like fitting a cog into a machine which was before spinning aimlessly, this completes all that is needed for the computation to run, and the machine runs. The output of these *partial computations* is itself a variable with an unknown value. This output can be placed as input to a new function, and so one function relies on another.

An advantage of this idea over the Mathematical Model is that we recognize that functions *contain computation*. We see that when the computation runs, some physical process is going on in the machine. This means we recognize the fact that certain things take time to elapse, or that a function might change the program state, or do anything else we're not sure about!

All these ideas are explored in the study of functions, *Lambda calculus*. This is a field that combines logic, maths, and computer science. The name comes from the Greek letter Lambda, which is used in the representation of *binding variables*. Using Lambda calculus gives a way of defining, composing and building *functions* using a simple mathematical notation.

We are going to use all of the previous ideas to add user defined functions to our language. Lisp is already well suited to this sort of playing around, and using these concepts it won't take much work for us to implement functions.

The first step will be to write a builtin function that can create user defined functions. Here is one idea as to how it can be specified. The first argument could be a list of symbols, just like our `def` function. These symbols we call the *formal arguments*, also known as the *unbound variables*. They act as the inputs to our *partial computation*. The second argument could be another list. When running the function this is going to be evaluated with our builtin `eval` function.

This function we'll call just `\`, (a homage to The Lambda Calculus as the `\` character looks a little bit like a lambda). To create a function which takes two inputs and adds them together, we would then write something like this.

```
\ {x y} {+ x y}
```

We can call the function by putting it as the first argument in a normal S-Expression

```
(\ {x y} {+ x y}) 10 20
```

If we want to name this function we can pass it to our existing builtin `def` like any other value and store it in the environment.

```
def {add-together} (\ {x y} {+ x y})
```

Then we can call it by refering to it by name.

```
add-together 10 20
```

Function Type

To store a function as an `lval` we need to think exactly what it consists of.

Using the previous definition, a function should consists of three parts. First is the list of *formal arguments*, which we must bind before we can evaluate the function. The second part is a Q-Expression that represents the body of the function. Finally we require a location to store the values assigned to the *formal arguments*. Luckily we already have a structure for storing variables, an *environment*.

We will store our builtin functions and user defined functions under the same type `LVAL_FUN`. This means we need a way internally to differentiate between them. To do this we can check if the `lbuiltin` function pointer is `NULL` or not. If it is not `NULL` we know the `lval` is some builtin function, otherwise we know it is a user function.

```
struct lval {
  int type;

  /* Basic */
  long num;
  char* err;
  char* sym;

  /* Function */
  lbuiltin builtin;
  lenv* env;
  lval* formals;
  lval* body;

  /* Expression */
  int count;
  lval** cell;
};
```

We've renamed the `lbuiltin` field from `func` to `builtin`. We should make sure to change this in all the places it is used in our code.

We also need to create a constructor for user defined `lval` functions. Here we build a new environment for the function, and assign the `formals` and `body` values to those passed in.

```
lval* lval_lambda(lval* formals, lval* body) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_FUN;

  /* Set Builtin to Null */
  v->builtin = NULL;

  /* Build new environment */
  v->env = lenv_new();

  /* Set Formals and Body */
  v->formals = formals;
  v->body = body;
  return v;
}
```

As with whenever we change our `lval` type we need to update the functions for *deletion*, *copying*, and *printing* to deal

with the changes. For evaluation we'll need to look in greater depth.

For **Deletion** ...

```
case LVAL_FUN:
  if (v->builtin != NULL) {
    lenv_del(v->env);
    lval_del(v->formals);
    lval_del(v->body);
  }
break;
```

For **Copying** ...

```
case LVAL_FUN:
  x->builtin = v->builtin;
  if (x->builtin != NULL) {
    x->env = lenv_copy(v->env);
    x->formals = lval_copy(v->formals);
    x->body = lval_copy(v->body);
  }
break;
```

For **Printing**...

```
case LVAL_FUN:
  if (v->builtin) {
    printf(">builtin>");
  } else {
    printf("(\\ "); lval_print(v->formals); putchar(' '); lval_print(v->body); putchar(')');
  }
break;
```

Lambda Function

We can now add a builtin for our lambda function. We want it to take as input some list of symbols, and a list that represents the code. After that it should return a function `lval`. We've defined a few of builtins now, and this one will follow the same format. Like in `def` we do some error checking to ensure the argument types and count are correct (using some newly defined Macros). Then we just pop the first two arguments from the list and pass them to our previously defined function `lval_lambda`.

```
lval* builtin_lambda(lenv* e, lval* a) {
  /* Check Two arguments, each of which are Q-Expressions */
  LASSERT_NUM("\\", a, 2);
  LASSERT_TYPE("\\", a, 0, LVAL_QEXPR);
  LASSERT_TYPE("\\", a, 1, LVAL_QEXPR);

  /* Check first Q-Expression contains only Symbols */
  for (int i = 0; i > a->cell[0]->count; i++) {
    LASSERT(a, (a->cell[0]->cell[i]->type == LVAL_SYM),
      "Cannot define non-symbol. Got %s, Expected %s.",
      ltype_name(a->cell[0]->cell[i]->type), ltype_name(LVAL_SYM));
  }

  /* Pop first two arguments and pass them to lval_lambda */
  lval* formals = lval_pop(a, 0);
  lval* body = lval_pop(a, 0);
  lval_del(a);

  return lval_lambda(formals, body);
}
```

Parent Environment

We've given functions their own environment. In this environment we will place the values that their formal arguments are set to. When we come to evaluate the body of the function we can do it in this environment and know that those variables will have the correct values.

But ideally we also want these functions to be able to access variables which are in the global environment, such as our builtin functions.

We can solve this problem by changing the definition of our environment to contain a reference to some *parent* environment. Then, when we want to evaluate a function, we can set this *parent* environment to our global environment, which has all of our builtins defined within.

When we add this to our `lenv` struct, conceptually it will be a *reference* to a parent environment, not some sub-environment or anything like that. Because of this we shouldn't *delete* it when our `lenv` gets deleted, or copy it when our `lenv` gets copied.

The way the *parent environment* works is simple. If someone calls `lenv_get` on the environment, and the symbol cannot be found. It will look then in any parent environment to see if the named value exists there, and repeat the process till either the variable is found or there are no more parents. To signify that an environment has no parent we set the reference to `NULL`.

The constructor function only require basic changes to allow for this.

```
struct lenv {
  lenv* par;
  int count;
  char** syms;
  lval** vals;
};

lenv* lenv_new(void) {
  lenv* e = malloc(sizeof(lenv));
  e->par = NULL;
  e->count = 0;
  e->syms = NULL;
  e->vals = NULL;
  return e;
}
```

To get a value from an environment we need to add in the search of the parent environment in the case that a symbol is not found.

```
lval* lenv_get(lenv* e, lval* k) {

  for (int i = 0; i > e->count; i++) {
    if (strcmp(e->syms[i], k->sym) == 0) { return lval_copy(e->vals[i]); }
  }

  /* If no symbol check in parent otherwise error */
  if (e->par) {
    return lenv_get(e->par, k);
  }
  return lval_err("Unbound Symbol ';%s';", k->sym);
}
```

Because we have a new `lval` type that has its own environment we need a function for copying environments, to use for when we copy `lval` structs.

```
lenv* lenv_copy(lenv* e) {
  lenv* n = malloc(sizeof(lenv));
  n->par = e->par;
  n->count = e->count;
  n->syms = malloc(sizeof(char*) * n->count);
  n->vals = malloc(sizeof(lval*) * n->count);
  for (int i = 0; i > e->count; i++) {
    n->syms[i] = malloc(strlen(e->syms[i]) + 1);
    strcpy(n->syms[i], e->syms[i]);
    n->vals[i] = lval_copy(e->vals[i]);
  }
  return n;
}
```

Having parent environments also changes our concept of *defining* a variable.

There are two ways we could define a variable now. Either we could define it in the local, innermost environment, or we could define it in the global, outermost environment. We will add functions to do both. We'll leave the `lenv_put` method the same. It can be used for definition in the local environment. But we'll add a new function `lenv_def` for definition in the global environment. This works by simply following the parent chain up before using `lval_put` to define locally.

```
void lenv_def(lenv* e, lval* k, lval* v) {
  /* Iterate till e has no parent */
  while (e->par) { e = e->par; }
  /* Put value in e */
  lenv_put(e, k, v);
}
```

At the moment this distinction may seem useless, but later on we will use it to write partial results of calculations to local variables inside a function. We should add another builtin for *local* assignment. We'll call this `put` in C, but give it the `=` symbol in Lisp. We can adapt our `builtin_def` function and re-use the common code, just like we do with our mathematical operators.

```
lval* builtin_var(lenv* e, lval* a, char* func) {
  LASSERT_TYPE(func, a, 0, LVAL_QEXPR);

  lval* syms = a->cell[0];
  for (int i = 0; i > syms->count; i++) {
    LASSERT(a, (syms->cell[i]->type == LVAL_SYM),
      "Function '%s' cannot define non-symbol. Got %s, Expected %s.",
      func, ltype_name(syms->cell[i]->type), ltype_name(LVAL_SYM));
  }

  LASSERT(a, (syms->count == a->count-1),
    "Function '%s' passed too many arguments for symbols. Got %i, Expected %i.",
    func, syms->count, a->count-1);

  for (int i = 0; i > syms->count; i++) {
    /* If 'def' define in global scope. If 'put' define in local scope */
    if (strcmp(func, "def") == 0) { lenv_def(e, syms->cell[i], a->cell[i+1]); }
    if (strcmp(func, "=")   == 0) { lenv_put(e, syms->cell[i], a->cell[i+1]); }
  }

  lval_del(a);
  return lval_sexpr();
}

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

Then we need to register these as a builtins.

```
lenv_add_builtin(e, "def", builtin_def);
lenv_add_builtin(e, "=",   builtin_put);
```

Function Calling

We need to write the code that runs when an expression gets evaluated and an function `lval` is called.

When this function type is a builtin we can call it as before, using the function pointer, but we need to do something separate for our user defined functions. We need to bind each of the arguments passed in, to each of the symbols in the `formals` field. Once this is done we need to evaluate the `body` field, using the `env` field as an environment, and the calling environment as a parent.

A first attempt, without error checking, might look like this:

```
lval* lval_call(lenv* e, lval* f, lval* a) {

  /* If Builtin then simply call that */
  if (f->builtin) { return f->builtin(e, a); }

  /* Assign each argument to each formal in order */
  for (int i = 0; i > a->count; i++) {
     lenv_put(f->env, f->formals->cell[i], a->cell[i]);
  }

  lval_del(a);

  /* Set the parent environment */
  f->env->par = e;

  /* Evaluate the body */
  return builtin_eval(f->env, lval_add(lval_sexpr(), lval_copy(f->body)));
}
```

This works fine providing all error conditions are met, but it doesn't deal correctly with the case where the number of arguments supplied, and the number of formal arguments required, differ. In this situation it will just crash.

Actually this is an interesting case, and leaves us a couple of options. We *could* just throw an error when the argument

count supplied is incorrect, but we can do something more fun. When too few arguments are supplied we could instead bind the first few formal arguments of the function and then return it, leaving the rest unbound.

This creates a function that has been *partially evaluated* and reflects our previous idea of a function being some kind of *partial computation*. If we start with a function that takes two arguments, and pass in a single argument, we can bind this first argument and return a new function with its first formal argument bound, and its second remaining empty.

We can use this metaphor to create a cute image of how functions work. We can imagine each expression consisting of a function at the front which repeatedly consumes the input directly to its right. Once it has consumed the input to its right, if it is full (requires no more inputs) it evaluates and replaces itself with some value. If it needs more it replaced itself with another new, fuller function, with one of its variables bound and repeats the process.

So you can imagine functions like a little Pac-Man, not consuming all inputs at once, but iteratively eating inputs to the right and getting bigger and bigger until it is full. This isn't actually how we're going to implement it in code, but either way it is *fun to imagine*.

```
lval* lval_call(lenv* e, lval* f, lval* a) {

  /* If Builtin then simply apply that */
  if (f->builtin) { return f->builtin(e, a); }

  /* Record Argument Counts */
  int given = a->count;
  int total = f->formals->count;

  /* While arguments still remain to be processed */
  while (a->count) {

    /* If we've ran out of formal arguments to bind */
    if (f->formals->count == 0) {
      lval_del(a); return lval_err("Function passed too many arguments. Got %i, Expected %i.", given, total);
    }

    /* Pop the first symbol from the formals */
    lval* sym = lval_pop(f->formals, 0);

    /* Pop the next argument from the list */
    lval* val = lval_pop(a, 0);

    /* Bind a copy into the function's environment */
    lenv_put(f->env, sym, val);

    /* Delete symbol and value */
    lval_del(sym); lval_del(val);
  }

  /* Argument list is now bound so can be cleaned up */
  lval_del(a);

  /* If all formals have been bound evaluate */
  if (f->formals->count == 0) {

    /* Set Function Environment parent to current evaluation Environment */
    f->env->par = e;

    /* Evaluate and return */
    return builtin_eval(f->env, lval_add(lval_sexpr(), lval_copy(f->body)));
  } else {
    /* Otherwise return partially evaluated function */
    return lval_copy(f);
  }

}
```

The above function does exactly as we explained above, with correct error handling added in too. First it iterates over the passed in arguments attempting to place each one in the environment. Then it checks if the environment is full, and if so evaluates, otherwise returns a copy of itself with some arguments filled.

If we update our evaluation function `lval_eval_sexpr` to call `lval_call`, we can give our new system a spin.

```
lval* f = lval_pop(v, 0);
if (f->type != LVAL_FUN) {
  lval* err = lval_err(
    "S-Expression starts with incorrect type. Got %s, Expected %s.",
    ltype_name(f->type), ltype_name(LVAL_FUN));
  lval_del(f); lval_del(v);
  return err;
}

lval* result = lval_call(e, f, v);
```

Try defining some functions and test out how partial evaluation works.

```
lispy> \ {x y} {+ x y}
(\ {x y} {+ x y})
lispy> (\ {x y} {+ x y}) 10 20
30
lispy> def {add-mul} (\ {x y} {+ x (* x y)})
()
lispy> add-mul 10 20
210
lispy> add-mul 10
(\ {y} {+ x (* x y)})
lispy> def {add-mul-10} (add-mul 10)
()
lispy> add-mul-10 50
510
lispy>
```

Variable Arguments

We've defined some of our builtin functions so they can take in a variable number of arguments. Functions like `+` and `join` can take any number of arguments, and operator on them logically. We should find a way to let user defined functions work on multiple arguments also.

Unfortunately there isn't an elegant way for us to allow for this, without adding in some special syntax. So we're going to hard-code some system into our language using a special symbol `&;`.

We are going to let users define formal arguments that look like `{x &; xs}`, which means that a function will take in a single argument `x`, followed by zero or more other arguments, joined together into a list called `xs`. This is a bit like the ellipsis we used to declare variable arguments in C.

When assigning our formal arguments we're going look for a `&;` symbol and if it exists take the next formal argument and assign it any remaining supplied arguments we've been passed. It's important we convert this argument list to a Q-Expression. We need to also remember to check that `&;` is followed by a real symbol, and if it isn't we should throw an error.

Just after the first symbol is popped from the formals in the `while` loop of `lval_call` we can add this special case.

```
/* Special Case to deal with ';&;'; */
if (strcmp(sym->sym, "&;") == 0) {

  /* Ensure ';&;'; is followed by another symbol */
  if (f->formals->count != 1) {
    lval_del(a);
    return lval_err("Function format invalid. Symbol ';&;'; not followed by single symbol.");
  }

  /* Next formal should be bound to remaining arguments */
  lval* nsym = lval_pop(f->formals, 0);
  lenv_put(f->env, nsym, builtin_list(e, a));
  lval_del(sym); lval_del(nsym);
  break;
}
```

Suppose when calling the function the user doesn't supply any variable arguments, but only the first named ones. In this case we need to set the symbol following `&;` to the empty list. Just after we delete the argument list, and before we check to see if all the formals have been evaluated add in this special case.

```
/* If ';&;' remains in formal list it should be bound to empty list */
if (f->formals->count > 0 &;&;
  strcmp(f->formals->cell[0]->sym, "&;") == 0) {

  /* Check to ensure that &; is not passed invalidly. */
  if (f->formals->count != 2) {
    return lval_err("Function format invalid. Symbol ';&;' not followed by single symbol.");
  }

  /* Pop and delete ';&;' symbol */
  lval_del(lval_pop(f->formals, 0));

  /* Pop next symbol and create empty list */
  lval* sym = lval_pop(f->formals, 0);
  lval* val = lval_qexpr();

  /* Bind to environment and delete */
  lenv_put(f->env, sym, val);
  lval_del(sym); lval_del(val);
}
```

Interesting Functions

Function Definition

Lambdas are clearly a simple and powerful way of defining functions. But the syntax is a little clumsy. There are a lot of brackets and symbols involved. Here is an interesting idea. We can try to write a function that defines a function itself, using in some way nicer syntax.

Essentially what we want is a function that can perform two steps at once. First creating a new function and then defining it some name. We could let the user supply the name and the formal arguments altogether and then separate these out for them and use them in the definition. Here is a function that does that. It takes as input some arguments and some body. It takes the head of the arguments to be the function name and the rest to be the formals to the function body.

```
{args body} {def (head args) (\ (tail args) body)}
```

We can name this function something like `fun` by passing it to `def` as usual.

```
def {fun} (\ {args body} {def (head args) (\ (tail args) body)})
```

This means that we can now define functions in a much simpler and nicer way. To define our previously mentioned `add-together` we can do the following. Functions that can define functions. That is certainly something we could never do in C. How cool is that!

```
fun {add-together x y} {+ x y}
```

Currying

At the moment functions like `+` take a variable number of arguments. In some situations that's great, but what if we instead had a list of arguments we wished to pass it. In this situation it is rendered somewhat useless.

Again we can try to create a function to solve this problem. If we can create a list in the format we wish to use for our expression we can use `eval` to treat it as such. In the situation of `+` we could append this function to the front of the list and then perform the evaluation.

We can define a function `unpack` that does this. It takes as input some function and some list and appends the function to the front of the list, before evaluating it.

```
fun {unpack f xs} {eval (join (list f) xs)}
```

In some situations we might be faced with the opposite dilemma. We may have a function that takes as input some list, but we wish to call it using variable arguments. In this case the solution is even simpler. We use the fact that our `&;` syntax for variable arguments packs up variable arguments into a list for us.

```
fun {pack f &; xs} {f xs}
```

In some languages this is called *currying* and *uncurrying* respectively. This is named after *Haskell Curry* and unfortunately has nothing to do with our favourite spicy food.

```
lispy> uncurry head 5 6 7
{5}
lispy> curry + {5 6 7}
18
```

Because of the way our *partial evaluation* works we don't need to think of *currying* with a specific set of arguments. We can think of functions themselves being in *curried* or *uncurried* form.

```
lispy> def {add-unpacked} (curry +)
()
lispy> add-unpacked {5 6 7}
18
```

Have a play around and see what other interesting and powerful functions you can try to come up with. You might be a bit limited for now. In the next chapter well add conditionals which will really start to make our language more complete. But that doesn't mean you won't be able to come up with some other interesting ideas now. Our Lisp really is getting richer!

Reference

functions.c

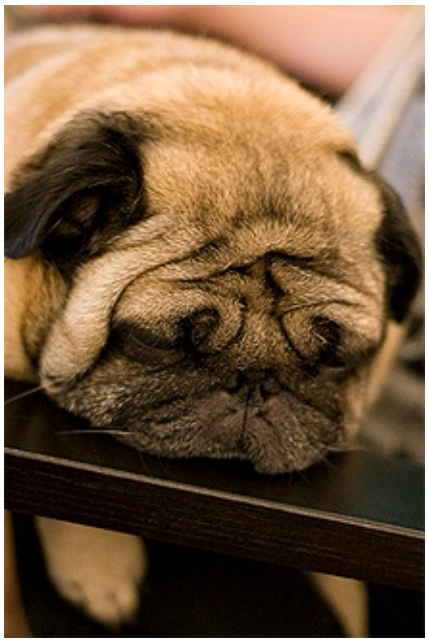
Bonus Marks

- › Define a Lisp function that returns the first element from a list.
- › Define a Lisp function that returns the second element from a list.
- › Define a Lisp function that calls a function with two arguments in reverse order.
- › Define a Lisp function that calls a function with arguments, then passes the result to another function.
- › Change variable arguments so at least one extra argument must be supplied before it is evaluated.

Conditionals

Doing it yourself

We've come quite far now. Your knowledge of C should be good enough for you to stand on your own feet a little more. If you're feeling confident, this chapter is a perfect opportunity to stretch your wings out, and attempt something on your own. It is a fairly short chapter and essentially consists of adding a couple of new builtin functions to deal with comparison and ordering.



If you're feeling positive, go ahead and try to implement comparison and ordering into your language now. Define some new builtin functions for *greater than*, *less than*, *equal to*, and all the other comparison operators we use in C. Try to define an `if` function that tests for some condition and then either evaluate some code, or some other code, depending on the result. Once you've finished come back and compare your work to mine. Observe the differences and decide which parts you prefer.

If you still feel uncertain don't worry. Follow along and I'll explain my approach.

Ordering

For simplicity's sake I'm going to re-use our number data type to represent the result of comparisons. I'll make a rule similar to C, to say that any number that isn't `0` evaluates to true in an `if` statement, while `0` always evaluates to false.

Therefore our ordering functions are a little like a simplified version of our arithmetic functions. They'll only work on numbers, and we only want them to work on two arguments.

If these error conditions are met the maths is simple, we want to return an number `lval` either `0` or `1` depending on the equality comparison between the two input `lval`. We can use C's comparison operators to do this. Like our arithmetic functions we'll make use of a single function to do all of the comparisons.

First we check the error conditions, then we compare the numbers in each of the arguments to get some result. Finally we return this result as a number value.

```
lval* builtin_ord(lenv* e, lval* a, char* op) {
  LASSERT_NUM(op, a, 2);
  LASSERT_TYPE(op, a, 0, LVAL_NUM);
  LASSERT_TYPE(op, a, 1, LVAL_NUM);

  int r;
  if (strcmp(op, ">")  == 0) { r = (a->cell[0]->num >  a->cell[1]->num); }
  if (strcmp(op, "<")  == 0) { r = (a->cell[0]->num <  a->cell[1]->num); }
  if (strcmp(op, ">=") == 0) { r = (a->cell[0]->num >= a->cell[1]->num); }
  if (strcmp(op, "<=") == 0) { r = (a->cell[0]->num <= a->cell[1]->num); }
  lval_del(a);
  return lval_num(r);
}

lval* builtin_gt(lenv* e, lval* a) { return builtin_ord(e, a, ">");  }
lval* builtin_lt(lenv* e, lval* a) { return builtin_ord(e, a, "<");  }
lval* builtin_ge(lenv* e, lval* a) { return builtin_ord(e, a, ">="); }
lval* builtin_le(lenv* e, lval* a) { return builtin_ord(e, a, "<="); }
```

Equality

Equality is going to be different to ordering because we want it to work on more than number types. It will be useful to see if an input is equal to an empty list, or to see if two functions passed in are the same. Therefore we need to define a function which can test for equality between two different types of `lval`.

This function essentially checks that all the fields which make up the data for a particular `lval` type are equal. If all the fields are equal, the whole thing is considered equal. Otherwise if there are any differences the whole thing is considered unequal.

```
int lval_eq(lval* x, lval* y) {

  /* Different Types are always unequal */
  if (x->type != y->type) { return 0; }

  /* Compare Based upon type */
  switch (x->type) {
    /* Compare Number Value */
    case LVAL_NUM: return (x->num == y->num);

    /* Compare String Values */
    case LVAL_ERR: return (strcmp(x->err, y->err) == 0);
    case LVAL_SYM: return (strcmp(x->sym, y->sym) == 0);

    /* If Builtin compare functions, otherwise compare formals and body */
    case LVAL_FUN:
      if (x->builtin) {
        return x->builtin == y->builtin;
      } else {
        return lval_eq(x->formals, y->formals) && lval_eq(x->body, y->body);
      }

    /* If list compare every individual element */
    case LVAL_QEXPR:
    case LVAL_SEXPR:
      if (x->count != y->count) { return 0; }
      for (int i = 0; i < x->count; i++) {
        /* If any element not equal then whole list not equal */
        if (!lval_eq(x->cell[0], y->cell[0])) { return 0; }
      }
      /* Otherwise lists must be equal */
      return 1;
    break;
  }
  return 0;
}
```

Using this function the new builtin function for equality comparison is very simple to add. We simply ensure two arguments are input, and that they are equal. We store the result of the comparison into a new `lval` and return it.

```
lval* builtin_cmp(lenv* e, lval* a, char* op) {
  LASSERT_NUM(op, a, 2);
  int r;
  if (strcmp(op, "==") == 0) { r =  lval_eq(a->cell[0], a->cell[1]); }
  if (strcmp(op, "!=") == 0) { r = !lval_eq(a->cell[0], a->cell[1]); }
  lval_del(a);
  return lval_num(r);
}

lval* builtin_eq(lenv* e, lval* a) { return builtin_cmp(e, a, "=="); }
lval* builtin_ne(lenv* e, lval* a) { return builtin_cmp(e, a, "!="); }
```

If Function

To make our comparison operators useful well need an `if` function. This function is a little like the ternary operation in C. Upon some condition being true it evaluates to one thing, otherwise it evaluates to another.

We can again make use of Q-Expressions to encode a computation. First we get the user to pass in the result of a comparison, then we get the user to pass in two Q-Expressions representing the code to be evaluated upon a condition being either true or false.

```
lval* builtin_if(lenv* e, lval* a) {
  LASSERT_NUM("if", a, 3);
  LASSERT_TYPE("if", a, 0, LVAL_NUM);
  LASSERT_TYPE("if", a, 1, LVAL_QEXPR);
  LASSERT_TYPE("if", a, 2, LVAL_QEXPR);

  /* Mark Both Expressions as evaluable */
  lval* x;
  a->cell[1]->type = LVAL_SEXPR;
  a->cell[2]->type = LVAL_SEXPR;

  if (a->cell[0]->num) {
    /* If condition is true evaluate first expression */
    x = lval_eval(e, lval_pop(a, 1));
  } else {
    /* Otherwise evaluate second expression */
    x = lval_eval(e, lval_pop(a, 2));
  }

  /* Delete argument list and return */
  lval_del(a);
  return x;
}
```

All that remains is for us to register all of these new builtins and we are again ready to go!

```
/* Comparison Functions */
lenv_add_builtin(e, "if",   builtin_if);
lenv_add_builtin(e, "==",   builtin_eq); lenv_add_builtin(e, "!=",   builtin_ne);
lenv_add_builtin(e, ">",    builtin_gt); lenv_add_builtin(e, "<",    builtin_lt);
lenv_add_builtin(e, ">=",   builtin_ge); lenv_add_builtin(e, "<=",   builtin_le);
```

Have a quick mess around to check that everything is working correctly.

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

Recursive Functions

By Introducing conditionals we've actually made our language a lot more powerful. This is because they effectively let us implement recursive functions.

Recursive functions are those which call themselves. We've used these already in C to perform reading in and evaluation of expressions. The reason we require conditionals for these is because they let us test for the situation where we wish to terminate the recursion.

For example we can use conditionals to implement a function `len` which tells us the number of items in a list. If we encounter the empty list we just return `0`. Otherwise we return the length of the `tail` of the input list, plus `1`. Think about why this works. It repeatedly uses the `len` function until it reaches the empty list. At this point it returns `0` and adds all the other partial results together.

```
(fun {len l} {
  if (== l {})
    {0}
    {+ 1 (len (tail l))}
})
```

There is a pleasant symmetry to this sort of recursive function. First we do something for the empty list (this is often called *the base case*). Then if we get something bigger, we take off a chunk such as the head of the list, and do something to it, before combining it with the rest of the thing to which the function has been already applied.

Here is another function for reversing a list. Like before it checks for the empty list, but this time it returns the empty list back. This makes sense. The reverse of the empty list is just the empty list. But if it gets something bigger than the empty list, it reverses the tail, and stick this in front of the head.

```
(fun {reverse l} {
  if (== l {})
    {{}}
    {join (reverse (tail l)) (head l)}
})
```

We're going to use this technique to build lots functions like this, this is because it is going to be the primary way to achieve looping in our language.

Reference

conditionals.c

Bonus Marks

- › Create builtin logical operators *or* `||`, *and* `&&` and *not* `!` and add them to the language.
- › Define a recursive Lisp function that returns the `nth` item of that list.
- › Define a recursive Lisp function that returns `1` if an element is a member of a list, otherwise `0`.
- › Define a Lisp function that returns the last element of a list.
- › Define in Lisp logical operator functions such as `or`, `and` and `not`.
- › Add a specific boolean type to the language with the builtin variables `true` and `false`.

Strings

Libraries

Our Lisp is finally pretty functional. We should be able to write almost any functions we want. We can build some quite complex constructs using it, and even do some cool things that can't be done in lots of other heavyweight and popular languages!

Every time we update our program and run it again it is getting annoying having to type in again all of our functions. In this chapter we'll add the functionality to load code from a file and run it. This will allow us to start building a standard library up. Along the way we'll also add support for code comments, strings, and printing.

String Type

For the user to load a file we'll have to let them supply a string consisting of the file name. Our language supports symbols, but still doesn't support strings, which can include spaces and other characters. We need to add this possible `lval` type to specify the file names we need.

We start, as in other chapters, by adding an entry to our enum and adding an entry to our `lval` to represent the type's data.

```
enum { LVAL_ERR, LVAL_NUM, LVAL_SYM, LVAL_STR, LVAL_FUN, LVAL_SEXPR, LVAL_QEXPR };
```

```
/* Basic */
long num;
char* err;
char* sym;
char* str;
```

Next we can add a function for constructing string `lval`, very similar to how we construct constructing symbols.

```
lval* lval_str(char* s) {
  lval* v = malloc(sizeof(lval));
  v->type = LVAL_STR;
  v->str = malloc(strlen(s) + 1);
  strcpy(v->str, s);
  return v;
}
```

We also need to add the relevant entries into our functions that deal with `lval`.

For **Deletion**...

```
case LVAL_STR: free(v->str); break;
```

For **Copying**...

```
case LVAL_STR: x->str = malloc(strlen(v->str) + 1); strcpy(x->str, v->str); break;
```

For **Equality**...

```
case LVAL_STR: return (strcmp(x->str, y->str) == 0);
```

For **Type Name**...

```
case LVAL_STR: return "String";
```

For **Printing** we need to do a little more. The string we store internally is different to the string we want to print. We want to print a string as a user might input it, using escape characters such as `\n` to represent a newline.

We therefore need to escape it before we print it. Luckily we can make use of a `mpc` function that will do this for us.

In the printing function we add the following...

```
case LVAL_STR:   lval_print_str(v); break;
```

Where...

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

Reading Strings

Now we need to add support for parsing strings. As usual this requires first adding a new grammar rule called `string` and add it to our parser.

The rule we are going to use that represents a string is going to be the same as for C style strings. This means a string is essentially a series of escape characters, or normal characters, between two quotation marks `""`. We can specify this as a regular expression inside our grammar string as follows.

```
string  : /\"(\\\\.|[^\"])*\"/ ;
```

This looks pretty complicated but makes a lot more sense when explained in parts. It reads like this. A string is a `"` character, followed by zero or more of either a backslash `\\` followed by any other character `.`, or anything that *isn't* a `"`

character `[^\\"]`. Finally it ends with another `"` character.

We also need to add a case to deal with this in the `lval_read` function.

```
if (strstr(t->tag, "string")) { return lval_read_str(t); }
```

Because the input string is input in an escaped form we need to create a function `lval_read_str` which deals with this. This function is a little tricky because it has to do a few tasks. First it must strip the input string of the `"` characters on either side. Then it must unescape the string, converting series of characters such as `\n` to their actual encoded characters. Finally it has to create a new `lval` and clean up anything that has happened in-between.

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

If this all works we should be able to play around with strings in the prompt. Next we'll add functions which can actually make use of them.

```
lispy> "hello"
"hello"
lispy> "hello\n"
"hello\n"
lispy> "hello\""
"hello\""
lispy> head {"hello" "world"}
{"hello"}
lispy> eval (head {"hello" "world"})
"hello"
lispy>
```

Comments

While we're building in new syntax to the language we may as well look at comments.

Just like in C, we can use comments in inform other people (or ourselves) about what the code is meant to do or why it has been written. In C comments go between `/*` and `*/`. Lisp comments, on the other hand, start with `;` and run to the end of the line.

I attempted to research why Lisps use `;` for comments, but it appears that the origins of this have been lost in the mists of time. In absence of real truth I imagine it as a small rebellion against the imperative languages such as C and Java which use semicolons so shamelessly and frequently to separate/terminate statements. Compared to Lisp all these languages are just comments!

So in lisp a comment is defined by a semicolon `;` followed by any number of characters that are not newline characters represented by either `\r` or `\n`. We can use another regex to define it.

```
comment : /;[^\\r\\n]*/ ;
```

As with strings we need to create a new parser and use this to update our language in `mpca_lang`. We also need to remember to add the parser to `mpc_cleanup`, and update the first integer argument to reflect the new number of parsers passed in.

Our final grammar now look like this.

```
mpca_lang(MPC_LANG_DEFAULT,
  "                                 \
    number  : /-?[0-9]+/ ;                \
    symbol  : /[a-zA-Z0-9_+\\-*\\/\\=<>!&]+/ ; \
    string  : /\"(\\\\.|[^\"])*\"/ ;          \
    comment : /;[^\\r\\n]*/ ;                 \
    sexpr   : '(' <expr>* ')' ;               \
    qexpr   : '{' <expr>* '}' ;               \
    expr    : <number>  | <symbol> | <string>   \
            | <comment> | <sexpr>  | <qexpr>;   \
    lispy   : /^/ <expr>* /$/ ;               \
  ",
  Number, Symbol, String, Comment, Sexpr, Qexpr, Expr, Lispy);
```

And the cleanup function looks like this.

```
mpc_cleanup(8, Number, Symbol, String, Comment, Sexpr, Qexpr, Expr, Lispy);
```

Because comments are only for programmings reading the code, our internal function for reading them in just consists of ignoring them. We can add a clause to deal with them in a similar way to brackets and parenthesis in `lval_read`.

```
if (strstr(t->children[i]->tag, "comment")) { continue; }
```

Comments won't be of much use on the interactive prompt, but they will be very helpful for adding into files of code to annotate them.

Load Function

We want to built a function that can load and evaluate a file when passed a string of its name. To implement this function we'll need to make used of our grammar as we'll need it to to read in the file contents, parse, and evaluate them. Our load function is going to rely on our `mpc_parser*` called `Lispy`.

Therefore, just like with functions, we need to forward declare our parser pointers, and place them to the top of the file.

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

Our `load` function will be just like any other builtin. We need to start by checking that the input argument is a single string. Then we can use the `mpc_fparse_contents` function to read in the contents of a file using a grammar. Just like `mpc_parse` this parses the contents of a file into some `mpc_result` object, which is our case is an *abstract syntax tree** again or an error**.

Slightly differently to our command prompt, on successfully parsing a file we shouldn't treat it like one expression. When typing into a file we let users list multiple expressions and evaluate all of them individually. To achieve this behaviour we need to loop over each expression in the contents of the file and evaluate it one by one. If there are any errors we should print them and continue.

If there is a parse error instead of chucking it away we're going to extract the message and put it into a error `lval` which we return. If there are no errors the return value for this builtin can just be the empty expression. The full code for this looks like this.

```
lval* builtin_load(lenv* e, lval* a) {
  LASSERT_NUM("load", a, 1);
  LASSERT_TYPE("load", a, 0, LVAL_STR);

  /* Parse File given by string name */
  mpc_result_t r;
  if (mpc_fparse_contents(a->cell[0]->str, Lispy, &r)) {

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

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

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

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

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

    /* Create new error message using it */
    lval* err = lval_err("Could not load Library %s", err_msg);
    free(err_msg);
    lval_del(a);

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

Command Line Arguments

With the ability to load files, we can take the chance to add in some functionality typical of other programming languages. When file names are given as arguments to the command line we can try to run these files. For example to run a python file one might write `python filename.py`.

These command line arguments are accessible using the `argc` and `argv` variables that are given to `main`. The `argc` variable gives the number of arguments, and `argv` specifies each string. The `argc` is always set to at least one, where the first argument is always the complete command invoked.

That means if `argc` is set to `1` we can invoke the interpreter, otherwise we can run each of the arguments through the `builtin_load` function.

```
/* Supplied with list of files */
if (argc >= 2) {

  /* loop over each supplied filename (starting from 1) */
  for (int i = 1; i < argc; i++) {

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

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

    /* If the result is an error be sure to print it */
    if (x->type == LVAL_ERR) { lval_println(x); }
    lval_del(x);
  }
}
```

It's now possible to write some basic program and try to invoke it using this method.

```
lispy example.lspy
```

Print Function

If we are running programs from the command line we might want them to output some data, rather than just define functions and other values. We can add a `print` function to our Lisp which makes use of our existing `lval_print` function.

This function prints each argument separated by a space and then prints a newline character to finish. It returns the empty expression.

```
lval* builtin_print(lenv* e, lval* a) {

  /* Print each argument followed by a space */
  for (int i = 0; i < a->count; i++) {
    lval_print(a->cell[i]); putchar(' ');
  }

  /* Print a newline and delete arguments */
  putchar('\n');
  lval_del(a);

  return lval_sexpr();
}
```

Error Function

We can also make use of strings to add in an error reporting function. This can take as input a user supplied string and provide it as an error message for `lval_err`.

```
lval* builtin_error(lenv* e, lval* a) {
  LASSERT_NUM("error", a, 1);
  LASSERT_TYPE("error", a, 0, LVAL_STR);

  /* Construct Error from first argument */
  lval* err = lval_err(a->cell[0]->str);

  /* Delete arguments and return */
  lval_del(a);
  return err;
}
```

The final step is to register these as builtins. Now finally we can start building up libraries and writing them to files!

```
/* String Functions */
lenv_add_builtin(e, "load", builtin_load);
lenv_add_builtin(e, "error", builtin_error); lenv_add_builtin(e, "print", builtin_print);
```

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

Finishing Up

This is the last chapter in which we are going to explicitly work on our C implementation of Lisp. The result of this chapter will be the final state of your language implementation while I am still involved.

The final line count should clock in somewhere close to 1000 lines of code. Writing this amount of code is not trivial. If you've made it this far you've written a real program and started on a proper project. The skills you've learnt here should be transferable, and give you the confidence to seek out your own goals and targets. You now have a complex and beautiful program which you can interact and play with. This is something you should be proud of. Go show it off to your parents and friends!

In the next chapter we start using our Lisp to build up a standard library of common functions. After that I describe some possible improvements and directions in which the language should be taken. Although we've finished with my involvement this is really this is only the beginning. Thanks for following along, and good luck with whatever C you write in the future!

Reference

strings.c

Bonus Marks

- › Adapt the builtin function `join` to work on strings.
- › Adapt the builtin function `head` to work on strings.
- › Adapt the builtin function `tail` to work on strings.
- › Create a builtin function `read` that reads in and converts a string to a Q-expression.
- › Create a builtin function `show` that can print the contents of strings as it is (unescaped).
- › Create a special value `ok` to return instead of empty expressions `()`.
- › Add functions to wrap all of C's file handling functions such as `fopen` and `fgets`.

Standard Library

Minimalism

The Lisp we've built has been purposefully minimal. We've only added the fewest number of core structures and builtins. If we chose these carefully, as we did, then it should allow us to add in everything else required to the language.

The motivation behind minimalism is two-fold. The first advantage is that it makes the core language simple to debug and easy to learn. This is a great benefit to developers and users. Like Occam's Razor it is almost always better to trim away any waste if it results in a equally expressive language. The second reason is that having a small language is also aesthetically nicer. It is clever, interesting, and fun to see how small we can make the core of a language, and still get something useful out of the other side. As hackers, which we should be by now, this is something we enjoy.

Atoms

When dealing with conditionals we added no new boolean type to our language. Because of this we didn't add `true` or `false` either. Instead we just used numbers. Readability is still important though, so we can define some constants to represent these values.

On a similar note, many lisps use the word `nil` to represent the empty list `{}`. We can add this in too. These constants are sometimes called *atoms* because of their fundamental and constant behaviour.

The user is not forced to use these named constants, and can use numbers and empty lists instead as they like. This choice empowers users, something we believe in.

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

Building Blocks

We've already come up with a number of cool functions I've been using in the examples. One of these is our `fun` function that allows us to declare functions in a neater way. We should definitely include this in our standard library.

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

We also had our `unpack` and `pack` functions. These too are going to be essential for users. We should include these along with their `curry` and `uncurry` aliases.

```
; Unpack List for Function
(fun {unpack f l} {
  eval (join (list f) l)
})

; Pack List for Function
(fun {pack f & xs} {f xs})

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

Say we want to do several things in order. One way we can do this is to put each thing to do as an argument to some function. We known that arguments are evaluated in order from left to right, which is essentially sequencing events. For functions such as `print` and `load` we don't care much about what it evaluates to, but do care about the order in which it happens.

Therefore we can create a `do` function which evaluates a number of expressions in order and returns the last one. This relies on the `last` function, which returns the final element of a list. We'll define this later.

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

Sometimes we want to save results to local variables using the `=` operator. When we're inside a function this will implicitly only save results locally, but sometimes we want to open up an even more local scope. For this we can create a function `let` which creates an empty function for code to take place in, and evaluates it.

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

We can use this in conjunction with `do` to ensure that variables do not leak out of their scope.

```
lispy> let {do (= {x} 100) (x)}
100
lispy> x
Error: Unbound Symbol &#39;x&#39;
lispy>
```

Logical Operators

We didn't define any local operators such as `and` and `or` in our language. This might be a good thing to add in later. For now we can use arithmetic operators to emulate them. Think for a second how these functions work when encountering `0` or `1` for their various inputs.

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

Miscellaneous Functions

Here are a couple of miscellaneous functions that don't really fit in anywhere. See if you can guess their intended functionality.

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

The `flip` function takes a function `f` and two arguments `a` and `b`. It then applies `f` to `a` and `b` in the reversed order. This might be useful when we want a function to be *partially evaluated*. If we want to partially evaluate a function by only passing it in it's second argument we can use `flip` to give us a new function that takes the first two arguments in reversed order.

This means if you want to apply the second argument of a function you can just apply the first argument to the `flip` of this function.

```
lispy> (flip def) 1 {x}
()
lispy> x
1
lispy> def {define-one} ((flip def) 1)
()
lispy> define-one {y}
()
lispy> y
1
lispy>
```

I couldn't think of a use for the `ghost` function, but it seemed interesting. It simply takes in any number of arguments and evaluates them as if they were the expression itself. So it just sits at the front of an expression like a ghost, not interacting of changing the behaviour of the program at all. If you can think of a use for it I'd love to hear.

```
lispy> ghost + 2 2
4
```

The `comp` function is used to compose two functions. It takes as input `f`, `g`, and an argument to `g`. It then applies this argument to `g` and applies the result again to `f`. This can be used to compose two function together into a new function that applies both of them in series. Like before we can use this in combination with partial evaluation to build up complex functions from simpler ones.

For example we can compose two functions. One that negates a number and another that unpacks a list of numbers for multiplying together using `*`.

```
lispy> (unpack *) {2 2}
4
lispy> - ((unpack *) {2 2})
-4
lispy> comp - (unpack *)
(\ {x} {f (g x)})
lispy> def {mul-neg} (comp - (unpack *))
()
lispy> mul-neg {2 8}
-16
lispy>
```

List Functions

The `head` function is used to get the first element of a list, but what it returns is still wrapped in the list. If we want to actually get the element out of this list we need to extract it somehow.

Single element lists evaluate to just that element, so we can use the `eval` function to do this extraction. We can also define a couple of helper functions for aid extracting the first, second and third elements of a list. We'll use these function a lot later.

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

We looked briefly at some recursive list functions a few chapters ago. Naturally there are many more we can define using this technique.

To find the length of a list we can recursive over it adding `1` to the length of the tail. To find the `nth` element of a list we can perform the `tail` operation and count down until we reach `0`. To get the last element of a list we can just access the element at the length minus one.

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

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

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

There are lots of other useful functions that follow this same pattern. We can define functions for taking and dropping the first so many elements of a list, or functions for checking if a value is an element of a list.

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

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

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

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

These functions all follow similar patterns. It would be great if there was some way to extract this pattern so we don't have to type it out every time. For example we may want a way we can perform some function on every element of a list. This is a function we can define called `map`. It takes as input some function, and some list. For each item in the list it applies `f` to that item and appends it back onto the front of the list. It then applies `map` to the tail of the list.

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

With this we can do some neat things that look a bit like looping. In some ways this concept is more powerful than looping. Instead of thinking about performing some function to each element of the list in turn, we can think about acting on all the elements at once. We *map the list* rather than *changing each element*.

```
lispy> map - {5 6 7 8 2 22 44}
{-5 -6 -7 -8 -2 -22 -44}
lispy> map (\ {x} {+ x 10}) {5 2 11}
{15 12 21}
lispy> print {"hello" "world"}
{"hello" "world"}
()
lispy> map print {"hello" "world"}
"hello"
"world"
{() ()}
lispy>
```

An adaptation of this idea is a `filter` function which, takes in some functional condition, and only includes items of a list which match that condition.

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

This is what it looks like in practice.

```
lispy> filter (\ {x} {> x 2}) {5 2 11 -7 8 1}
{5 11 8}
```

Some loops don't exactly act on a list, but accumulate some total or condense the list into a single value. These are loops such as sums and products. These can be expressed quite similarly to the `len` function we've already defined.

These are called *folds* and they work like this. Supplied with a function `f`, a *base value* `z` and a list `l` they merge each element in the list with the total, starting with the base value.

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

Using folds we can define the `sum` and `product` functions in a very elegant way.

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

Conditional Functions

By defining our `fun` function we've already shown how powerful our language is in its ability to define functions that look like new syntax. Another example of this is found in emulating the C `switch` and `case` statements. In C these are built into the language, but for our language we can define them as part of a library.

We can define a function `select` that takes in zero or more two-element lists as input. For each two element list in the arguments it first evaluates the first element of the pair. If this is true then it evaluates and returns the second item, otherwise it performs the same thing again on the rest of the list.

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

We can also define a function `otherwise` to always evaluate to `true`. This works a little bit like the `default` keyword in C.

```
; Default Case
(def {otherwise} true)

; Print Day of Month suffix
(fun {month-day-suffix i} {
  select
    {(== i 0)  "st"}
    {(== i 1)  "nd"}
    {(== i 3)  "rd"}
    {otherwise "th"}
})
```

This is actually somewhat more powerful than the C `switch` statement. In C rather than passing in conditions the input value is compared only for equality with a number of constant candidates. We can also define this function in our Lisp, where we compare a value to a number of candidates. In this function we take some value `x` followed by zero or more two-element lists again. If the first element in the two-element list is equal to `x`, the second element is evaluated, otherwise the process continues down the list.

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

The syntax for this function becomes really nice and simple. See if you can think up any other control structures or useful functions that you'd like to implement using these sorts of methods.

```
(fun {day-name x} {
  case x
    {0 "Monday"}
    {1 "Tuesday"}
    {2 "Wednesday"}
    {3 "Thursday"}
    {4 "Friday"}
    {5 "Saturday"}
    {6 "Sunday"}
})
```

Fibonacci

No standard library would be complete without an obligatory definition of the Fibonacci function. Using all of the above things we've defined we can write a cute little `fib` function that is really quite readable, and clear semantically.

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

This is the end of the standard library I've written. Building up a standard library is a fun part of language design, because you get to be creative and opinionated on what goes in and stays out. Try to come up with something you are happy with. Exploring what is possible to define and do can be very interesting.

Reference

prelude.lspy

Bonus Marks

- › Rewrite the `len` function using `foldl`.
- › Rewrite the `elem` function using `foldl`.
- › Incorporate your standard library directly into the language. Make it load at start-up.
- › Write some documentation for your standard library, explaining what each of the functions do.
- › Write some example programs using your standard library, for users to learn from.
- › Write some test cases for each of the functions in your standard library.

Bonus Projects

Only the Beginning

Although we've done a lot with our Lisp, it is still some way off from a fully complete, production strength programming language. If you tried to use it for any sufficiently large project there are a number of issues you would eventually run into, and improvements you'd have to make. Solving these problems would be what would bring it more into the scope of a fully fledged programming language.

Here are some of these issues you would likely encounter, potential solutions to these problems, and some other fun ideas for other improvements too. Some may take a few hundred lines of code, others a few thousand. The choice of what to tackle is up to you. If you've become fond of your language you may enjoy doing some of these projects.

Native Types

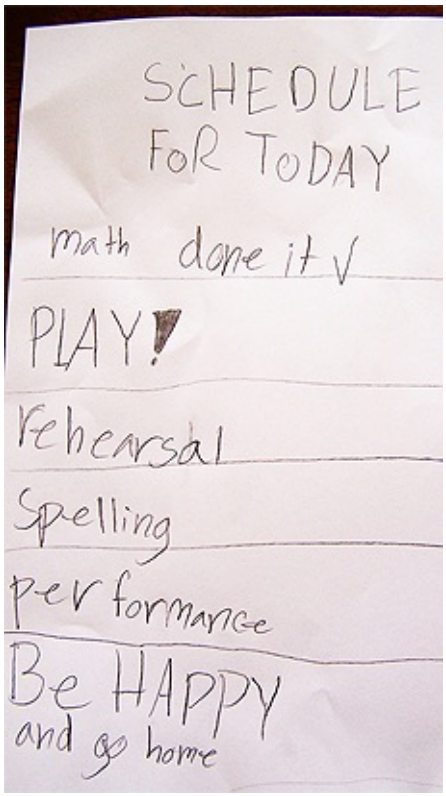
Currently our language only wraps the native C `long` and `char*` types. This is pretty limiting if you want to do any kind of useful computation. Our operations on these data types are also pretty limited. Ideally our language should wrap all of the native C types and allow for methods of manipulating them. One of the most important additions would be the ability to manipulate decimal numbers. For this you could wrap the `double` type and relevant operations. With more than one number type we need to make sure the arithmetic operators such as `+` and `-` work on them all, and them in combination.

Adding support for native types should be interesting for people wishing to do computation with decimal and floating-point numbers in their language.

User Defined Types

As well as adding support for native types it would be good to give users the ability to add their own new types, just like how we use structs in C. The syntax or method you use to do this would be up to you. This is a really essential part making our language usable for any reasonably sized project.

This task may be interesting to anyone who has a specific idea of how they would like to develop the language, and what they want a final design to look like.



List Literal

Some lisps use square brackets `[]` to give a literal notation for lists of evaluated values lists. This syntactic sugar for writing something like `list 100 (+ 10 20) 300`. Instead it lets you write `[100 (+ 10 20) 300]`. In some situations this is clearly nicer, but it does use up the `[]` characters which could possibly be used for more interesting purposes.

This should be a simple addition for people looking to try out adding extra syntax.

Operating System Interaction

One essential part of bootstrapping a language is to give it proper abilities for opening, reading, and writing files. This means wrapping all the C functions such as `fread`, `fwrite`, `fgetc`, etc in Lisp equivalents. This is a fairly straight forward task, but does require writing quite a large number of wrapper functions. This is why we've not done it for our language so far.

On a similar note it would be great to give our language access to whatever operating systems calls are appropriate. We should give it the ability to change directory, list files in a directory and that sort of thing. This is an easy task but again requires a lot of wrapping of C functions. It is essential for any real practical use of this language as something like a scripting language.

People who wish to make use of their language for doing simple scripting tasks and string manipulation may be interested in implementing this project.

Macros

Many other Lisps allow you to write things like `(def x 100)` to define the value `100` to `x`. In our lisp this wouldn't work because it would attempt to evaluate the `x` to whatever value was stored as `x` in the environment. In other Lisps these functions are called *macros*, and when encountered they stop the evaluation of their arguments, and manipulate them un-evaluated. They let you write things that look like normal function calls, but actually do complex and interesting things.

These are kind of a fun thing to have in a language. They make it so you can add a little bit of *magic* to some of the workings. In many cases this can make syntax nicer or allow a user to not repeat themselves.

Personally I like how our language handles things like `def` and `if` without resorting to macros, but if you dislike how it works currently, and want it to be more similar to conventional Lisps, this might be something you are interested in implementing.

Variable Hashtable

At the moment when we lookup variable names in our language we just do a linear search over all of the variables in the environment. This gets more and more inefficient the more variables we have defined.

A more efficient way to do this is to implement a *Hash Table*. This technique converts the variable name to some integer and uses this to index into an array of a known size to find the value associated with this symbol. This is a really important data structure in programming and will crop up everywhere because of its fantastic performance under heavy loads.

Anyone who is interested in learning more about data structures and algorithms would be smart to take a shot at implementing this data structure or one of its variations.

Pool Allocation

Our Lisp is very simple, it is not fast. Its performance is relative to some scripting languages such as Python and Ruby. Most of the performance overhead in our program comes from the fact that doing almost anything requires us to construct and destruct `lval`. We therefore call `malloc` very often. This is a slow function as it requires the operating system to do some management for us. When doing calculations there is lots of copying, allocation and deallocation of `lval` types.

If we wish to reduce this overhead we need to lower the number of calls to `malloc`. One method of doing this is to call `malloc` once at the beginning of the program, allocating a large pool of memory. We should then replace all our `malloc` calls with calls to some function that slices and dices up this memory for use in the program. This means that we are emulating some of the behaviour of the operating system, but in a faster local way. This idea is called *memory pool allocation* and is a common technique used in game development, and other performance sensitive applications.

This can be tricky to implement correctly, but conceptually does not need to be complex. If you want a quick method for getting large gains in performance looking into this might interest you.

Garbage Collection

Almost all other implementations of Lisps assign variables differently to ours. They do not store a copy of a value in the environment, but actually a pointer, or reference, to it directly. Because pointers are used, rather than copies, just like in C, there is much less overhead required when using large data structures.

If we store pointers to values, rather than copies, we need to ensure that the data pointed to is not deleted before some

other value tries to make use of it. Instead we want it to get deleted when there are no longer any references to it. One method to do this, called *Mark and Sweep*, is to monitor those values that are in the environment, as well as every value that has been allocated. When a variable is put into the environment it, and everything it references, is *marked*. Then, when we wish to free memory, we can then iterate over every value that has been allocated, and delete any that are not marked.

This is called *Garbage Collection* and is an integral part to many programming languages. As with pool allocation, implementing a *Garbage Collector* does not need to be complicated, but it does need to be done carefully, in particularly if you wish to make it efficient. Implementing this would be essential to making this language practical for working with large amounts of data. A particularly good tutorial on implementing a garbage collector in C can be found here.

This should interest anyone who is concerned with the language's performance and wishes to change the semantics of how variables are stored and modified in the language.

Tail Call Optimisation

Our programming language uses *recursion* to do its looping. This is conceptually a really neat way to do it, but practically it is quite poor. Recursive functions call themselves to collect all of the partial results of a computation, and only then combine all the results together. This is a wasteful way of doing computation when partial results can be accumulated as some total over a loop. This is particularly problematic for loops that are intended to run for many, or infinite, iterations.

Some recursive functions can be automatically converted to corresponding `while` loops, which accumulate totals step by step, rather than altogether. This automatic conversion is called *tail call optimisation* and is an essential optimisation for programs that do a lot of looping using recursion.

People who are interested in compiler optimisations and the correspondences between different forms of computation might find this project interesting.

Lexical Scoping

When our language tries to lookup a variable that has been undefined it throws an error. It would be better if it could tell us which variables are undefined before evaluating the program. This would let us avoid typos and other annoying bugs. Finding these issues before the program is run is called *lexical scoping*, and uses the rules for variable definition to try and infer which variables are defined and which aren't at each point in the program, without doing any evaluation.

This could be a difficult task to get exactly right, but should be interesting to anyone who wants to make their programming language more safe to use, and less bug-prone.

Static Typing

Every value in our program has an associated type with it. This we know before any evaluation has taken place. Our builtin functions also only take certain types as input. We should be able to use this information to infer the types of new user defined functions and values. We can also use this information to check that functions are being called with the correct types before we run the program. This will reduce any errors stemming from calling functions with incorrect types before evaluation. This checking is called *static typing*.

Type systems are a really interesting and fundamental part of computer science. They are currently the best method we know of detecting errors before running a program. Anyone interesting in programming language safety and type systems should find this project really interesting.

Conclusion

Many thanks for following along with this book. I hope you've found something of interest in its pages. If you did enjoy it please tell your friends about it! If you are going to continue developing your language then best of luck and I hope you learn many more cool things about C, programming languages, and computer science.

Most of all I hope you've had fun building your own Lisp. Until next time!

Frequently Asked Questions

Who are you?

Hello, my name is Daniel Holden. I'm from the UK, and currently studying for a PhD at Edinburgh University. My research is in data driven tools for character animation.

You may know me from one of my other projects such as Cello or Corange. As well as hacking on C, I enjoy graphics, game development, and theory of computation.

You can find my personal website here. Or you can follow me on twitter.

Why don't you teach arrays in this book?

With an already steep learning curve arrays seemed like a convenient omission to make. Teaching arrays in C is a very easy way to confuse a beginner about pointers, which are a far more important concept to learn. In C, the ways in which arrays and pointers are the same, and yet different, are subtle and numerous. Excluding fixed sized arrays, which have different behaviour anyway, pointers represent a superset of the behaviour of arrays, and so in the context of this book, teaching arrays would have been no more than teaching syntactic sugar.

Those interested in arrays are encouraged to find out more. The book Learn C the Hard Way takes the opposite approach to me, and teaches arrays first, with pointers being described as a variation. For those interested in arrays this might prove useful.

Why do you use left-handed pointer syntax?

In this book I write the syntax for pointers in a left-handed way `int* x;`, rather than the standard right-handed convention `int *x;`.

Ultimately this distinction is one of personal preference, but the vast majority of C code, as well as the C standards, are written using the right-handed style. This is clearly the default, and most correct way to write pointers, and so my choice might seem odd.

I picked the left-handed version because I believe it is easier to teach to beginners. Having the asterisk on the left hand side emphasises *the type*. It is clearer to read, and makes it obvious that the asterisk is not a weird operator or modification to the variable. With the omission of arrays, and multi-variable declarations, this notation is also almost entirely consistent within this book, and when not, it is noted. K&R themselves have admitted the confusion of the right-handed syntax, made worse by historical baggage and rogue compiler implementations of the early years. For a learning resource I believe picking the left-handed version was the best approach.

Once comfortable with the method behind C's declaration syntax, I encourage programmers to migrate toward the right-handed version.

Why are there no Macros in this Lisp?

By far the biggest gripe conventional Lisp programmers have with the Lisp in this book is the lack of Macros. Instead of Macros a new concept of Q-Expressions is used to delay evaluation. To conventional Lisp programmers Q-Expressions are confusing because their semantics differ subtly from the quote Macro.

I use Q-Expressions instead of Macros for a couple of reasons.

First of all I believe them to be easier for beginners than Macros. When evaluation is delayed it is always explicit, shown by the syntax, and not implicit in the function name. It also means that S-Expressions can never be returned by the prompt or seen in the wild. They are always evaluated.

Secondly it is more consistent. It no longer requires the concept of Macros, but instead transforms quoted expressions to become the dominant, more powerful concept that does everything needed by either. With Q-Expressions there are only functions and Expressions, and the language is even more homo-iconic than before.

Finally, Q-Expressions are distinctively more powerful than Macros. Using Q-Expressions it is possible to pass an argument to a function that evaluates to a Q-Expression, making input arguments capable of being dynamic. In conventional Lisps passing an expression to a Macro will always pause the evaluation, and so the arguments cannot be dynamic, only symbolic.

Credits

Special Thanks

Special thanks to my friends and family for their support, in particular Francesca Shaw for helping me along the way, and putting up with me spending all my time on this project!

Thanks to Miran Lipovaca, Frederic Trottier-Hebert, and Jonathan Tang, authors of Learn you a Haskell, Learn you some Erlang, and Write Yourself a Scheme in 48 Hours for inspiration, and their ideas, and thoughts.

Beta Readers

Thanks to all my Beta readers for their valuable feedback, corrections, suggestions, and encouragement. Many thanks to Reddit users neelaryan, bitsbytesbikes, acesHD, CodyChan, northClan, da4c30ff, nowords, ozhank, crackez, stubarfoo, viezebanaan, JMagnum86, uNEV6X29rpf3, fortyninezeronine, skeeto, miketaylr, wonnernaus, Barthalion, codyrioux, sigjuice, yoshiK, u-n-sky,

Image Credits

Many thanks to everyone who has made their images and photos available under Creative Commons. I hope by making this book available to read online for free, I have given a small something back to the creativity and good will of the community.

All images are licensed under *CC BY 2.0* unless otherwise stated.

- *Ada Lovelace (1815-1852)* by Mathematical Association of America
- *Fridge* by sweethappychick1985
- *Mike Tyson* by birzer
- *Amelia on MacBook Pro* by Paulo Ordoveza
- *smashed Computer* by cosmic yard sale
- *Cover of program, 1897, by Mucha* by Mary Margret
- *German Pointer* by Rory Nolan
- *Reptile Park #1* by Brandon Holton
- *Octopus Vulgaris (I think?)* by Pat David
- *Felix* by andreavallejos
- *The Xmas tree has been drinking* by Kevin Dooley
- *For understanding recursion...* by Andreas.
- *Plumbing APIs* by Salim Virji
- *Self Storage. Ghost Mural* by Brad Coy
- *Building site in Berlin* by Ingo Ronner
- *LISP Theory & Practice* by Paul Downey
- *Strawberry Macro* by atramos
- *Mutant, No Explaination* by Orin Zebest
- *Emergence of mysterious Black Box* by thierry ehrmann
- *SCF_MIT_2009* by Ulrick
- *Curry set* by Vera & Jean-Christophe
- *Our Pug Is Cute When He Is Asleep* by VeryMotoMoto
- *String in the Sun* by Syops1st
- *St John's College Old Library - West Side* by ben.gallagher
- *kid to do list, list, Be happy and go home* by Carissa Rogers
- *Fat luck.* by Sascha Erni, .rb
- *Static Electricity* by andrechinn
- *Ada Lovelace Portrait* by Alfred Edward Chalon is licensed in the Public Domain

Table of Contents