CMSC330 Notes

Cliff

Contents

Chapter 1

Intro

Hello There

General Kenobi

I took this course many moons ago and so now I'm making notes based on what I remember from the course and my own experience playing around with programming languages. That being said, I take a pretty holistic approach to this course. That is, I assume that you have a good understanding of the previous classes you needed to get here and we will talk about how what you learn here relates to what you have learned previously.

1.1 What is a language?

After pulling out my HESP140 notes, I can say with some confidence that a formal definition for language can be simply put as a system of communication. However, after pulling out my philosophy notes[1], I want to say that language is anything that transfers what goes on 'in our mind' to 'out of our mind.' I'm sure more important people would disagree, but eh.

What I am trying to get at is the fact that we all have thoughts and feelings, and ultimately no one knows what goes on in our heads until we express or share what we are thinking and feeling [2] To be put even more succinctly: a language is a medium used to express ourselves. And in my experience, programming languages are no different. However it's not that simple.

1.2 How do we use language?

So now that we answered what a language is, **we can now focus on one part of what this course wants to make sure you understand: how to express ourselves with a language**. To answer this question, we need to learn some new words: **semantics** and **syntax**. **semantics refers to the meaning of sentences/languages** while **Syntax refers to the structure of the language**[3]. Consider the following:

- The snow is white
- schnee ist weiß
- Precipitation comprised of ice cystrals under the temperture of 0° C reflects all wavelengths of light from 400nm to 700 nm.

I would argue that all 3 sentences have the same meaning. That is, the semantics of the sentences are the same. Programming languages are the same. Consider the following:

[1] I would recommend taking philosophy of language with Alexander Williams

[2] Thomas Nagel has a fun little paper called "What is it like to be a bat" that says no matter how much we know about bats and no matter how hard we imagine what echolocation would be like, we could not experience how a bat navigates.

[3] Some would say this sounds like grammar. This is partially true since grammar is a subset of syntax

```
\\java
x % 2 == 0 ? System.out.println("Even"):System.out.println("Odd");

\* C *\
if (x % 2 == 0){
    printf("Even\n");
}else{
    printf("Odd\n");
}
```

The semantics of these two programs are the same, despite them looking different. This is where syntax comes into play. Since syntax deals with the rules of what is valid or not, let us take an even smaller example:

```
\\java
System.out.println("Hello, World!");

\*C*\
printf("Hello, World!");
```

Syntax deals with what is valid rules to create a sentence in a language. If we tried to write the first line in a C program, the compiler will yell at us, and had we tried to write some C code in Java, the compiler will be yell at us again. That is to express the same thing, in one language requires one thing, and in another something else. We will eventually talk about grammar, but for now just keep in mind the idea of syntax and semantics. **An array will always be an array, but the code you use to make one will differ from language to language**. That said, most languages are Turing complete (which we will also discuss later), so basically any program you make in one language, can be made in a different one which raises the following question(s).

1.3 Why so many languages? Why use one over another?

As we hopefully all know, computers only really know is machine code. But we as programmers don't really know or play around with bytecode. We use other languages which are easier to write with because they have shortcuts or macros that cover the hard and tedious stuff. Consider the following assembly code:

```
func:
    push    ebp
    mov     ebp,esp
    ; code
    mov     esp, ebp
    pop     ebp
```

This particular example represents the stack frame that is added to the stack whenever a function is called. It would be terrible if we had to do this everytime we wanted to call a function, so if we can replace the aforementioned assembly with something nice:

```
    func();
```

That is most languages have some way to implement or represent a function call (typically means adding parenthesises after the function name). **The idea of having special shortcuts in a language is the basis for the second point this course finds important: language features**. Different languages features is why there are so many languages and why you way want to use one language instead of another.

1.4 Language Features

Let us consider the following Java code:

```
int sum = 0
for(int i = 0; i < 10; i++){
    sum += i
}
```

Now consider the following LSIP code which does the exact same thing:

```
(defun sum (s a) (if (= a 0) s (sum (+ s a) (- a 1))))
(sum 0 10)
```

Now these two code segments do the same thing, but notice that LISP's doesn't seem fun or as straight forward. We will discuss later in the course why that is, and other fun thinga about this, but for now **you just need to know that there is no such thing as iterative structures in LISP**. LISP is purely recursive (and we will learn this with OCaml) and so it has a different way to express what to do. Throughout this course, you will see this idea over and over: **some languages have certain ways to expressing things that others do not**. When talking about these features, I will try to highlight why the feature is useful and more importantly how this feature is implemented in the backend, and how you could incorporate it in other languages. **By the end of this class you should be able to make your own or at least modify programming languages to have features from other languages**. An example of this with languages and ideas you already should know would be knowing how to implement object oriented inheritance in C (hint: void pointers and structs).

1.5 Conclusion

This course focuses on a few things

- Recognizing language features and analyzing their effects on problem solving
- Imperative and functional programming
- Computational abilities of regular, context-free and Turing-complete languages
- Deriving meaning from a language
- Creating proofs about the correctness of a program
- Designing languages and implementing their evaluation

Chapter 2

Ruby

> A regretful honey maker could be called a rue-bee
>
> Kliff

This is a programming language chapter so it has two (2) main things: talk about some properties that the Ruby programming language has and the syntax the language has. If you want to code along, all you need is a working version of Ruby and a text editor. You can check to see if you have Ruby installed by running `ruby -version`. At the time of writing, I am using Ruby 3.0.4. You can also download an interactive ruby shell called `irb`.

2.1 Introduction

Unlike the previous languages you have seen in 131/132 and 216, Ruby does not use a compiler so no machine code is generated. This means that compile-time checks do not exist. This basically means that every check or error is done during run-time. I'll expand more on this in a bit but for now, lets write our first Ruby program.

```
1 # hello_world.rb
2 puts "Hello World"
```

Despite this being very simple, we already learned five (5) things.

- Single line comments are started with the pound or hashtag symbol
- no semicolons to denote end of statement
- `puts` is used to print things out to stdout (`print` if you don't want a newline at the end
- Parenthesis are technically optional when calling functions (but good style says to only leave out if there are no arguments or if the function is `puts`, `require`, or `include`)
- ruby file name conventions are `lowercase_and_underscore.rb`
- Strings exists in the language (most languages do, but some do not)

Now to run our ruby program we can just use

```
1 ruby hello_world.rb
```

Congrats, you have just made your first program in Ruby!

2.2 Typing

Now we just said that Ruby does not use compile-time checks and all checks and errors are done at run-time. Let's see this in action.

```
1  # program1.rb
2  variable1 = 5
3  variable1 = "hello"
4  variable2 = 4
5  variable3 = variable1 + variable2
6  puts varable3
```

Here is a simple program that sets some variables, adds two things together, and then prints the results. This looks weird, but first lets run and see what happens, and then we can look at the syntax.

```
ruby program1.rb
hw.rb:5:in `+': no implicit conversion of Integer into String (TypeError)
    from hw.rb:4:in `<main>'
```

Looks like we have an error! Seems like `variable1` and `variable2` have different types so we can't use `'+'` on them. Which is interesting for 2 main reasons

- We did not get an error on line 3 when we change `variable1` from an `Integer` value to a `String` value
- there is no automatic conversion of integers to strings like we saw in Java

The first point is really important, but before we talk about it, let's just modify our code so that it runs without any errors by deleting line 3.

```
7  # program1-1.rb
8  variable1 = 5
9  variable2 = 4
10 variable3 = variable1 + variable2
11 puts varable3
```

Now if we run our code we get a different error. We get **`'undefined local variable or method 'varable3' for main:Object (NameError)'`**. Notice that because we check everything during run-time, this error was not picked up until ruby was about to execute it. **Many bugs from beginner Ruby programmers is due to misspelled variable names**. Here is a more visual demonstration of run-time erring.

```
12 # program1-2.rb
13 variable1 = 5
14 variable2 = 4
15 variable3 = variable1 + variable2
16 print variable1
17 print " + "
18 print variable2
19 print " = " # gross. We will learn to convert later
20 puts varable3   # still misspelled
```

If we run the above code, we get **`'5 + 4 = '`** printed out and then we get the same **`'NameError'`** as before. A error free program would look like

```
21 # program1-3.rb
22 variable1 = 5
23 variable2 = 4
24 variable3 = variable1 + variable2
25 puts variable3
```

Now that we fixed this issue, we can talk about why we didn't get an error on line 3 in `program1.rb`.

Notice that in the above code we set values to variables but we didn't define the type like we did in Java and C. This is because Ruby uses **Dynamic type checking**. Dynamic type checking is a form of type checking which is typically contrasted to **Static type checking**. **Type checking** is an action that is used for a **Type system**, which determines how a language assigns a type to a variable. Ultimately: How does a a language know if a variable is an `int` or a `pointer`? It does so by type checking.

For the most part, you probably only used static type checking since both C and Java are statically typed. **Static typing** means that the type of a variable, construct, function, etc is known at compile time. Should we then use a type in an incorrect manner (as we did in `program1.rb`, then the compiler will raise an error and compilation will be aborted. Contrasted to **Dynamic typing** which means that the type is only calculated at run-time. Consider the following:

```
1 # err.c
2 void main(){
3     int x;
4     x = "hello"
5 }
```

Should we try to compile this code with the `-Werror`[1] compile flag, we will get the following: **`'error: assignment to 'int' from 'char *' makes integer from pointer without a cast'`**. This is because during compilation, the compiler marks the x variable as having a type of int yet is being assigned a pointer.

Now how did gcc know that there was a type issue here? The first conclusion would be that we declared x as an `int` explicitly on line 3. This is called **manifest or explicit typing** where we explicitly declare the type of any variable we create. This is in contrast to **latent or implicit typing** where we don't have to do this as we saw in ruby. It is important to note: **manifest typing is not the same as static typing**. We will see this in OCaml as the language is statically typed, but uses latent typing for its variables.

Back to Dynamic type checking, let's look at the following:

```
1 # checking.rb
2 def add(a,b)
3     puts a + b
4 end
5
6 add(1,2)
7 add("hello"," world")
```

To begin, this is how you create a function in Ruby. Functions begin with the `def` keyword and end with the end keyword. We will go into ruby code examples later so for now just know this is a function that takes two arguments and then prints the the result of `a + b`. Since Ruby is dynamically typed, we don't assign types to the parameters a and b until we run our code, and not until we actually use the values. This allows us to call `add` with both `Integers` and `Strings`. Much like before it is important to note that **latent typing is not the same as dynamic typing**.

In any case, you may be wondering Ruby knows the types of variables if we don't explicitly declare their types. The process of deciding a type for an expression is called **Type inference** and we will go more in depth with this in the OCaml section, but there are many ways that type inference can be done, and in fact you already saw one way with our `err.c` program. We did not say that `"hello"` was a `char *` yet gcc knew because of the syntax of the datatype. The same holds true for ruby. Integers are numbers without decimal points. Floats are numbers with decimal points. Strings are anything put in quotes.[2] You can check this with the `.class` method. Did I mention that Ruby is object oriented?

2.3 Object Oriented Programming

You should be familiar with Object Oriented Programming (OOP) because of Java, however, unlike Java, everything is an object in Ruby. Lets test this out.

```
1 # oop.rb
2 puts 3.class
3 puts "Hello".class
4 puts 4.5.class
```

[1] Canonically it will just raise an error and still compile

[2] We will see how ruby does this when we talk about parsing. In particular Project 4

You can see that everything, including primitives, are object oriented. Also notice that like java, when calling an object's methods, we use the dot syntax. That means that earlier when we called `a + b` in `add.rb`, it was actually doing something like `a.+(b)`. Don't believe me? Consider the following:

```
1 # program2.rb
2 m = 3.methods
3 puts methods.include?(:+)
4 puts 3.+(4)
```

The Line 2 just gets the methods which the object **'3'** has as an array. The Line 3 will then print out **'true'** because we are asking if the array of methods includes the **':+'** method. This is actually a symbol, but we will talk about that later. We can then call the **'+'** method on 3 adding it to 4 and we get **'7'** as output. Pretty weird right?

Other properties of OOP also exist in Ruby. As we saw before, objects have methods, and this is the primary way that objects interact with each other. Recall that Objects are instances of Classes so each Object has it's own state. This also means that all values are references to objects (so be careful how you check to see if two values are equal). Additionally, Ruby has an inheritance structure similar to Java. In Ruby, all classes are derived from the `Object` class.

```
1 # oop.rb
2 puts 3.class
3 puts 3.class.ancestors
4
5 a = "hello"
6 b = a
7 puts a.equal?(b)        #true
8 puts a.equal?("hello")  #false
9 puts "hello" == "hello" #true
```

Object oriented programming in Ruby also means that we need some sort of value to represent the absence of an object. In java it was called **'null'**, in Ruby, we call it **'nil'**. `nil` actually is an object itself and has methods which you can use.

```
1 # nil.rb
2 puts nil.methods
3 puts nil.to_s
```

2.3.1 Class Creation

Let's make our first class

```
1  # square.rb
2  class Square
3    def initialize(size)
4      @size = size
5    end
6
7    def area
8      @size*@size
9    end
10 end
11
12 s = Square.new(5)
13 puts s.area
```

There is a lot here, let's break it down.

- Lines 1 and 9 is the outline of the class. The name of the class is Square
- lines 2-4 is the Ruby equivalent to the constructor.

- lines 3 and 7 use `@size` which is a instance variable
- lines 6-8 is a instance method
- Line 11 is the instantiating of the newly made Square class
- line 12 is calling the instance method

The Equivalent Java code if below.

```java
// Square.java
public class Square{
  private int size;
  public initialize(size){
    this.size = size;
  }

  public int area(){
    return size*size;
  }
}

public static void main(String[] args){
  Square s = new Square(5);
  System.out.println(s.area);
}
```

Notice that the instance variable `size` is private which means if we wished to access it, we would need to make getters and setters. We could do this by adding the following

```ruby
# square-1.rb
class Square
# ...
    # getter
    def size
        @size
    end
    #setter
    def size=(s)
        @size = s
    end
end
# ...
```

This is annoying to do for each variable we have so Ruby actually has a built in function to help us: **`attr_accessor`** Consider the following:

```ruby
# square- 2.rb
class Square
    attr_accessor :size
    # ... same as before ...
end
s = Square.new(5)
puts s.area
s.size= 6
puts s.area
puts s.size
```

If you wanted to use `static` or class variables, you just prepend the variable name with **`'@@'`**. So if you wanted to count how many squares were made, you could do so like so:

```ruby
# square- 3.rb
class Square
    @@count = 0
    attr_accessor :size
    def initialize(size)
        @@count += 1
```

```
7           @size = size
8       end
9
10      def count
11          @@count
12      end
13      # ... same as before ...
14  end
15  s = Square.new(5)
16  s2 = Square.new(6)
17  puts s.count
```

For class or static variables you need to initialize them and write your own getters and setters. You can also make static methods by defining them in terms of the class. See below.

```
1   # counter.rb
2   class Counter
3       @@count = 0
4       def initialize()
5           @@count += 1
6       end
7
8       def Counter.counter
9           @@count
10      end
11  end
12  c = Counter.new
13  c1 = Counter.new
14  puts Counter.counter
```

One other thing you may have noticed is that we did not use the common **`return`** keyword. This is because Ruby will return whatever the last line in a function evaluates to. The following 3 methods all do the same thing:

```
1   # return.rb
2   def to_s1
3       s = "Hello"
4       return s
5   end
6
7   def to_s2
8       s = "Hello"
9       s
10  end
11
12  def to_s3
13      "Hello"
14  end
15   puts to_s1
16   puts to_s2
17   puts to_s3
```

Lastly, we stated earlier that classes are all derived from the `Object` class. This means we must have some form of inheritance. It acts much like Java, the syntax is just different.

```
1   # inheritance.rb
2   class Shape
3     def to_s
```

```
4       "I am a shape"
5     end
6   end
7
8   class Square < Shape
9     def to_s
10      super() + " and a square"
11    end
12  end
13  puts Square.new
```

The above created two classes, Shape and Square. A Square is a subclass of Shape and so it inherits all its methods. If we wish to override our parent's method, we can certainly do so as seen in lines 9-11. If we wish to refer to the parent's method we can do so using the super method. In fact we can override any method, but method overloading is not supported. We would get an error should be try to run

```
1   # overload-err.rb
2   class Square
3       def func1(x)
4           puts "func1"
5       end
6
7       def func1(x,y)
8           puts "func2"
9       end
10  end
11  Square.new.func1(2,3) #fine
12  Square.new.func1(2)   #error
```

But we could do something like the following

```
1   # override.rb
2   class Square
3       def func1(x)
4           puts "func1"
5       end
6
7       def func1(x)
8           puts "func2"
9       end
10  end
11  Square.new.func2(1)
```

This can lead to some pretty interesting behaviour where we can add things to existing Classes. In the following example, I am going to add a new method to the Integer class which just returns the double of the value.

```
1   # double.rb
2   puts 3.methods.include?(:double)
3
4   class Integer
5     def double
6       self + self
7     end
8   end
9
10  puts 3.methods.include?(:double)
11  puts 3.double
```

The `self` keyword is similar to java's `this`. It refers to the current object. We can also use this power of overriding methods to break ruby

```
1  # break.rb
2  class Integer
3    def +(x)
4      "Not Today"
5    end
6
7    def -(x)
8      self * x
9    end
10 end
11
12 puts 3+4
13 puts 3-4
```

If we run this in irb, it will crash, but if you save this as a file and run it, you get **`'Not Today'`** followed by **`'12'`**.

2.4 Code blocks

Okay, I'll be upfront. I lied earlier when I said everything is an object. Afaik there is only one feature of Ruby which is not object oriented: codeblocks. If you took a look at Section 2.6 you will know that we can create an Array the following way:

```
1  a = Array.new(3,"Item")
```

However, there is another way that you can initialize an array with a default value.

```
1  a = Array.new(3){"Item"}
```

This is an example of a codeblock. Codeblocks are typically surrounded in curly braces({}) but can also be surrounded with `do ... end`. Codeblocks are not objects so you cannot assign variables to them, nor can you call methods on them. Additionally you cannot pass them into functions as parameters, nor can you return a codeblock as a return value. However there is one important thing to take away from codeblocks: that we can treat code as data. Consider the following:

```
1  def func
2      if block_given?
3          yield
4      end
5  end
6  func1 {puts "hello"}
```

The `yield` keyword on line 3 tells ruby to pass control to the codeblock associated with the function. We can see this more clearly in the following example:

```
1  def func1
2      yield 5
3  end
4
5  def func2(i)
6      y = i+1
7      puts y
8  end
9
10 func1 {|i| puts i + 1}
11 func2(5)
```

Right off the bat, we should acknowledge that codeblocks can take in paramaters when yielded to, as seen on line 2. The syntax for accepting arguments can be shown on line 10, where you suround the arguments in pipbars (|). Anyway, the two

function calls on line 10 and 11 have similar behavior., The difference is that when using the codeblock, the parameter 5 is kept constant with the code being executed being variable on all calls of func1, whereas func(2) can take in a varaible parameter, but the code executed will always be the same. Let's see this more clearly:

```
# similar
func1 {|i| puts i + 1}
func2(5)

#not similar
func1{|i| puts i %2}
func2(6)

func1{|i| Array.new(i)}
func2(3)
```

If you squint, you can consider this similar to passing in a function pointer in C and then calling said function. Notice that control is passed to the codeblock on a yield and then returned when the codeblock finishes executing.

```
# codeblock not executed unless yield is called
def func3
    puts "hello"
end
func3 {puts "World"}

# control passed when yield is called
def func4
    yield 1,2
    yield 3,4
end
func4{|a,b| puts a + b}
```

Again, I will reiterate that codeblocks are not objects which means no passing them in as parameters or returning them.

```
# cannot do
def func5(i)
    yield i
    return { puts "hello"} #cannot do
end

func5({puts "hello"}) # error
```

THere is however a workaround. They are called Procs. Procs create this thing called a closure which we talk about in the OCaml sections. For now, just know that Procs allow us to store codeblocks inside an object.

```
p = Proc.new {puts "Hello"}
puts p.class
```

Because procs are objects and not a codeblock, you cannot yield to them, but there are methods you can call from a proc. To execute the code stored in a proc, we can use the `.call` method. Much like a codeblock, the body of a proc is not executed until the `call` method is called.

```
def func6(p)
    p.call
    p.call
end
p = Proc.new {puts "hello"}
func6(p)
```

Procs can also take multiple arguments, and afaik, unlike codeblocks, can be nested in eachother. For example

```ruby
def func7(x)
    p = Proc.new {|y| Proc.new {|z| x + y + z}}
    return p
end
a = func7(1)
b = a.call(2)
c = b.call(3)
puts c
```

Because Procs are objects, we can do some fun things:

```ruby
def map(arr,func)
    for value in arr
        puts func.call(value)
    end
end
map([1,2,3,4],Proc.new{|i| i + 1})

def execute(arr)
    for func in arr
        func.call
    end
end
funcs = [Proc.new{puts "hello"}, Proc.new{puts "Bye"}, Proc.new{puts "C you"}]
execute(funcs)
```

2.5 Modules

Now that we talked about the one thing that is not object oriented, let's go back and talk about one issue with object oriented programming in Ruby (and Java): inheritance restrictions. In these languages, we have the feature of inheritance, but we can only have one parent class which is not entirely feasible. In java we got around this with interfaces. In ruby, we can use modules.

Let's write our first module and then we can see how to go about using it:

```ruby
module Doubler
    def Doubler.base
        2
    end

    def double
        self + self
    end
end
```

This module has both a static and an instance method. The syntax for this module is similar to Ruby's Class creation. We create static methods with the `<Classname>.<method>` syntax and create instance methods using the common `def...end` keywords. There are a few things to note about Modules that make them different from classes:

- Modules cannot be instantiated
- Modules use the `module` keyword instead of the `class` keyword
- cannot be extended like a class

That's it. Pretty simple. We cannot extend modules but we can still overwrite them; But we are getting ahead of ourselves. Let's just first see how we use them.

```
class Integer
    include Doubler
end
puts 10.double
```

Here we are adding the `Doubler` module to the `Integer` class so any integer now has access to the `doubler` method. We can also do things like

```
puts Doubler.base
puts Doubler.class
puts Doubler.instance_methods
```

But because we cannot instantiate we cannot do

```
Doubler.new
Doubler.double
```

The only thing that may be confusing with Modules is when it comes to overwritten methods. Consider the following:

```
module M1
    def bye
        "Goodbye"
    end
end

module M2
    def bye
        "Bye"
    end
end

class C
    include M1
    include M2
end

puts C.new.bye
```

In this case, we load modules in the order we include them so M2 has the last instance of defining bye so M2's bye method will be called. Had we swapped the order:

```
class C
    include M2
    include M1
end
```

Then M1's bye method would be called instead. If we had an instance method in the C class, then we would call C's bye method.

```
class C
    include M2
    include M1

    def bye
        "C ya"
    end
end
```

Typically the order in which something is called is by first looking at self, then the self's modules, then the parent's instance methods, then the parent's modules, then the grandparent's instance methods, then the grandparent's modules, etc. We can see that here:

```ruby
module M1
    def bye
        "Goodbye"
    end
end

module M2
    def bye
        "Bye"
    end
end

class C
    include M1
end

class D < C
    include M2
end
puts D.new.bye
```

If you are ever unsure of the order, you can always use the `.ancestors` method.

```ruby
puts D.new.class.ancestors
# [D,M2,C,M1,Object,Kernel,BasicObject]
```

There is one important thing to note: once something is loaded, it will not be loaded again.

```ruby
class C
    include M2
    include M1
end

class D < C
    include M2
end
puts D.new.bye
```

Some Modules we have kinda seen before, namely the `Comparable` and `Enumerable` modules. Any Class that includes the `Comparable` module supports `<.>„<=,>=,=` operators. Classes that include the `Enumerable` allow things like `map` and `select`.

2.6 Data Types and Syntax

That is pretty much all you need to know about Ruby for this course for now. All that's left is to go over syntax of data types and other things.

2.6.1 Numbers

There are two common types of numbers: `Integers` and `Floats`. An `Integer` is a positive or negative integer value without a decimal point. A `Float` is a positive or negative value with a decimal point and at least one digit on either side of said point. When performing operations between the same types, the resulting value is the same type. When performing an operation which involves a `Float`, the resulting value is typically also a `Float`. For some reason, Ruby also allows you to use an underscore as a separator. Maybe for readability?

```ruby
# numbers.rb
-1 + 1 # addition between Integers
```

```ruby
6.5 % 1.2 # modulus between Floats

2. # not valid for floats
.1 # also not valid

3.0/2 # will result in a Float

1_000_000 == 1000000 # true
```

2.6.2 Stings and Symbols

Strings in ruby are anything in-between double or single quotes. Since things are Objects in Ruby, Strings follow structural equality, but not physical equality. You can nest single and double quotes if you want to print one or the other.

```ruby
# strings.rb
"String 1"
'String 1'
'String' == "String"      # true
"string".equal?("string") # false
```

Symbols on the other hand are special strings, but only one of each symbol exits meaning they are physically equal. Since they are physically equal, they are also structurally equal. A symbol can be any valid string but is written with a `':'` in the front. You can add quotes if you have a multi-word symbol. We have seen symbols when using `attr_accessor` and `.methods`.

```ruby
# symbol.rb
:"String 1"
:'String 1'
:"string".equal?(:"string") # true
:"string".equal?(:'string') # true
:"string".equal?(:string)   # true
```

2.6.3 Arrays

Arrays use the very common bracket syntax for both creation and indexing. Unlike in most languages, arrays can be heterogeneous which is nice. Ruby arrays also support dynamic sizing and set operations. Any value not initialized because `nil`. One important thing to note is that when dealing with n-Dimensional arrays, you must always have the previous dimension declared.

```ruby
# arr.rb
# creating
arr = []
arr = [1,2,3,4]
arr = [1,2.0,"hello"]

arr = Array.new(3) # [nil,nil,nil]
arr = Array.new(3,"a") # ["a","a","a"]

# indexing
a = [1,2,3,4]
puts a[0]   # 1
puts a[-1]  # 4

# dynamic sizing
arr = []
```

```ruby
arr[4] = 5
puts arr # [nil,nil,nil,nil,5]

# set stuff
a = [1,2,3,4,5]
b = [4,5,6,7,8]
puts a+b # [1,2,3,4,5,4,5,6,7,8]
puts a|b # [1,2,3,4,5,6,7,8]
puts a&b # [4,5]
puts a-b # [1,2,3]

#adding and removing
a = [1,2,3]
a.push(4)
puts a # [1,2,3,4]
a.pop
puts a # [1,2,3]
a.unshift(0)
puts a # [0,1,2,3]
a.shift
puts a # [1,2,3]
a.delete_at(1)
puts a # [1,3]
a.delete(3)
puts a # [1]

a2d = [][] # error
a2d = []
a2d[0] = []
puts a2d # [[]]

# you can also use a code block
a2d = Array.new(3){Array.new(3)} # create a 3x3 matrix
puts a2d #[[nil,nil,nil],[nil,nil,nil],[nil,nil,nil]]
```

Unlike some languages, Hashes are built into Ruby. This means you don't have to make your own hashing mechanism or hash function (though you should if you are doing this for security purposes). Ruby uses the common curly brace syntax for creation and the bracket syntax to index. If a key does not exist in the hash, it is automatically mapped to nil. You can change the default hash if you want. Hashes in ruby are very much like arrays, except instead of mapping numbers (or indexes) to values, you can map anything to anything. That is to say, keys do not have the be the same amongst each other and the same for values. Keys and values also do not have to have the same type. Like Arrays, when dealing with n-Dimensional arrays, you must always have the previous dimension declared.

```ruby
# arr.rb
# creating and indexing
h = {}
h = {"key" => :value,}
h = Hash.new
puts h['key'] # nil
h = Hash.new(:default)
puts h['key'] # :default

# adding
h = {}
h['key1'] = :value1
```

```ruby
puts h # {'key1'=>:value1}
h.delete('key1')
puts h # {}

# Multi-dimensional Hashes
h = {}
h[0] = {}
h[0][0] = 4
h2 = {}
h2[0][0] = 4 # error
```

2.6.4 Control Flow

The most simple version of control flow is the `if` statement. You should know what an `if` statement is by now so I won't discuss what they are or how they work. Instead lets talk about the bigger class of statements: control statements. Control statements control the flow of program execution; More specifically they alter which command comes next. There are several in Ruby: `if, while, for, until, do while` the main ones, but most people just use the first 3. For those that have a boolean check, **`'true'`** is anything that is not **`'false'`** or **`'nil'`**. **`'nil'`** is like null, it is used for initialized fields. however, **`'nil'`** is an object itself of the `NilClass`. **`'true'`** and **`'false'`** are also objects of `TrueClass` and `FalseClass` respectively. Note:**`FalseClass` and `NilClass` do not evaluate to false**. Consider the following:

```ruby
# conditional.rb
count = 1
while count >= 0
    if 3 > 4 then # then is optional
        puts hello
    elsif nil
        puts "nil is true"
    else
        if count == 0
            puts FalseClass == false
        end
        puts NilClass == false
    end
    count -= 1
end
```

You should run this code to see what happens but but here are 3 important things

- on line 5, `hello` is an undefined variable but Ruby never catches this. Since Ruby is dynamically typed, this bug goes unnoticed.
- instead of `elseif` or `else if`, ruby uses `elsif`. Why, I have no idea. This is a common bug
- the end keyword is commonly used whenever you would otherwise use } in other languages

You can read more at the Ruby Docs.

Chapter 3

Python

Python Comments are #great

Kliff

The Python programming language was created by Guido Von Rossum many moons ago around the 1980s with its first release in 1991. Since then, there have been various changes to the language, some of which are interesting and others which are not as interesting. It was named after the British comedy group Monty Python. For some buzzwords, we can say Python is a high-level, dynamically typed, garbage-collected, primarily imperative programming language. You may also hear the terms "duck typing", "scripting", and "interpreted". What do all of these weird terms mean? I am sure someone knows.

If you want to follow along through this chapter, you can run the files with **`python file.py`**. Alternatively, you can also use the repl (Read-Eval-Print-Loop). Just type `python` in your terminal. (At the time of writing, I am using python 3.8)

3.1 Introduction

Unlike the previous languages you have seen in 131/132 and 216, Python does not use a compiler, so no machine code is generated. This means that compile-time checks do not exist. This basically means that every check or error is done during run-time. I'll expand more on this in a bit, but for now, let's write our first Python program.

```
1 # hello_world.py
2 print("Hello World")
```

Despite this being very simple, we've already learned several things.

- Single-line comments are started with the pound or hashtag symbol
- no semicolons to denote the end of the statement (what does this imply?)
- `print` is used to print things out to stdout
- functions use parenthesis (weird we need to say this)
- python file entension is `.py`
- Strings exist in the language (most languages have them, but some do not)

Can you think of more?

Now to run our program we can just use

```
1 python hello_world.py
```

Congrats, you have just made your first program in Python!

3.2 Typing

Now we just said that Python does not use compile-time checks, and all checks and errors are done at run-time. Let's see this in action.

```
26 # program1.py
27 variable1 = 5
28 variable1 = "hello"
29 variable2 = 4
30 variable3 = variable1 + variable2
31 print(varable3)
```

Here is a simple program that sets some variables, adds two things together, and then prints the results. This looks weird, but first, let's run and see what happens, and then we can look at the syntax.

```
python program1.py
TypeError: can only concatenate str (not "int") to str
```

Looks like we have an error! Seems like `variable1` and `variable2` have different types, so we can't use `'+'` on them. This is interesting for 2 main reasons:

- We did not get an error on line 3 when we change `variable1` from an `Integer` value to a `String` value
- There is no automatic conversion of integers to strings as we saw in Java

The first point is really important, but before we talk about it, let's just modify our code so that it runs without any errors by deleting line 3.

```
32 # program1-1.py
33 variable1 = 5
34 variable2 = 4
35 variable3 = variable1 + variable2
36 print(varable3)
```

Now if we run our code, we get a different error. We get **`'NameError: name 'varable3' is not defined'`**. Notice that because we check everything during run-time, this error was not picked up until Python was about to execute it. **Many bugs from beginner (and experienced) Python programmers are due to misspelled variable names**. Here is a more visual demonstration of run-time erring.

```
37 # program1-2.py
38 variable1 = 5
39 variable2 = 4
40 variable3 = variable1 + variable2
41 print(variable1,end="") # no new line when printing. What can you infer about this?
42 print(" + ",end="")
43 print(variable2,end="")
44 print(" = ",end="") # gross. We will learn to convert later
45 print(varable3)   # still misspelled
```

If we run the above code, we get **`'5 + 4 = '`** printed out, and then we get the same **`'NameError'`** as before. An error-free program would look like

```
46 # program1-3.py
47 variable1 = 5
48 variable2 = 4
49 variable3 = variable1 + variable2
50 print(variable3)
```

Now that we fixed this issue, we can talk about why we didn't get an error on line 3 in `program1.rb`.

Notice that in the above code, we set values to variables, but we didn't define the type like we did in Java and C. This is because of Python's **type system**. A language's type system determines how data and variables can be used. Some key

words that may describe a type system could be dynamic vs static or manifest vs latent. You may also hear of type safety and strong vs weak. To determine how data and variables must be used, rules are made and enforced by what is typically called a type checker. Ultimately: How does a language know if a variable or piece of data is an `int` or a `pointer`? It does so by type-checking.

Python uses both dynamic and latent typing in its language. Dynamic type checking is a form of type checking which is typically contrasted to **Static type checking**. For the most part, you probably only used static type checking since both C and Java are statically typed. **Static typing** means that the type of a variable, construct, function, etc is known before the program runs. It can also be said that type checking is run at compile time. Should we then use a type in an incorrect manner (as we did in `program1.py`), then the compiler will raise an error and compilation will be aborted. Contrasted to **Dynamic typing** which means that the type is only calculated at run-time. (Since Python has no compiler, it cannot be statically typed). Consider the following:

```
1 # err.c
2 void main(){
3     int x;
4     x = "hello"
5 }
```

Should we try to compile this code with the `-Werror`[1] compile flag, we will get the following: **`'error: assignment to 'int' from 'char *' makes integer from pointer without a cast'`**. This is because during compilation, the compiler marks the x variable as having a type of int, yet it is being assigned a pointer.

Now, how did gcc know that there was a type issue here? The first conclusion would be that we declared `x` as an `int` explicitly on line 3. This is called **manifest or explicit typing**, where we explicitly declare the type of any variable we create. This is in contrast to **latent or implicit typing** where we don't have to do this as we saw in python. It is important to note: **manifest typing is not the same as static typing**. We will see this in OCaml as the language is statically typed, but uses latent typing for its variables.

Back to Dynamic type checking, let's look at the following:

```
1 # checking.py
2 def add(a,b):
3     print(a + b)
4
5 add(1,2)
6 add("hello"," world")
```

To begin, this is how you create a function in Python. Functions begin with the `def` keyword, and the body will always be indented. We will go into Python code examples later, so for now, just know this is a function that takes two arguments and then prints the result of `a + b`. Since Python is dynamically typed, types aren't assigned to the parameters `a` and `b` until we run our code, and not until the values are actually used. This allows us to call `add` with both `Integers` and `Strings`. Much like before, it is important to note that **latent typing is not the same as dynamic typing**.

In any case, you may be wondering how Python knows the types of variables if we don't explicitly declare their types. The process of deciding a type for an expression is called **Type inference** and we will go more in-depth with this in the OCaml section, but there are many ways that type inference can be done. In fact, you already saw one way with our `err.c` program. We did not say that `"hello"` was a `char *`, yet gcc knew because of the syntax of the datatype. The same holds true for Python. Integers are numbers without decimal points. Floats are numbers with decimal points. Strings are anything put in quotes. You can check this with the `type` function. For instance, try `type("hello")`.

3.3 Python Scoping

When we think of a scope, we think about where in the code can we use something, whether it be a variable or a piece of data. Consider the following:

[1] Canonically it will just raise an error and still compile

```
1  # scoping-1.py
2  a = 5
3  b = 20
4  def h(c):
5    d = True
6    e = 0
7    while d:
8      if c < b - c:
9        c = c + 1
10       e = e + 1
11     else:
12       d = False
13   return e
14
15 f = input("Enter a number: ")
16 print(h(int(f)) + a)
```

While this program has a ton of new syntax that we have not seen in python before, I believe we can take a look at this program, take a few notes, and make some hypotheses about some things we can and cannot do in python. You can check if your hypotheses are correct in the syntax part of this chapter [2].

In any case, we can ask ourselves: where can each variable be used? Do you believe you could use a from lines 2 - 16? Can c be used outside of lines 4 - 13? While these questions may have somewhat intuitive answers based on your past programming knowledge, you then have to consider: why does asking this question matter?

Consider the following:

```
1  # scoping-2.py
2  a = 5
3  b = 20
4  def h(a):
5    d = True
6    e = 0
7    while c:
8      if a < b - a:
9        a = a + 1
10       e = e + 1
11     else:
12       c = False
13   return e
14
15 f = input("Enter a number: ")
16 print(h(int(f)) + a)
```

Here, I changed a variable name, and we now need to consider the scope of the a variable and if this impacts the outcome of the program. In this case, no change to the program occurs, but what about something simple like the following?

```
1  # scoping-3.py
2  a = 5
3  def h():
4      print(a)
5  h()
```

[2] Whenever you learn a new language, a great way to pick it up quickly is to analyze a snippet of code and mark down what you think everything is doing. Then, make a hypothesis about what you think the code will do, and then run it. You will either affirm your assumptions and enforce your belief about how the language works, or you get something you did not expect. When this happens, this means there is a gap in your knowledge. You can now make assumptions about why the outcome occurred and what you didn't consider. The best thing to do here is to modify the code, make a new hypothesis, and continue this process to learn more. For example: I see that `True` and `False` exist and are capitalized here. I have two hypotheses: 1: booleans exist in the language. 2: You must capitalize true and false in the language. Now to test this: a simple `type(True)` tells you this is of type bool, which means python calls them bool and not boolean. Additionally, `type(true)` gives an error, which affirms the hypothesis that `bool`s must be capitalized.

Where can we use the a variable? If I modify this just a tad bit more:

```
# scoping-4.py
a = 5
def h():
    a = a + 1
    print(a)
h()
```

What does your intuition say? What does `python scoping-4.py` say? In this case, we get an error! Wild. Why is this?

Typically, global variables are used throughout the entire program, and if one part of the program can modify that variable, it would impact other parts of the program that use that variable. Thus, making global variables read-only is good practice. In this case, they are typically called global constants, or just constants. Since Python doesn't need a main function (but it can have one), any variable defined outside a function is global by default.

If we wish to modify a global variable, we can do so using the `global` keyword.

```
# scoping-5.py
a = 5
def h():
    global a
    a = a + 1
    print(a)
h()
```

So what happens if we have a local and global variable of the same name, like we did in `scoping-2.py`? Can we edit the global variable and not the local one?

```
# scoping-6.py
a = 5
def h():
    a = 5
    global a
    a = a + 1
    print(a)
h()
```

Here we get an error: a python function seems to only be able to deal with either a local variable or a global variable, not both. But notice the wording of the error message: `"name 'a' is used prior to global declaration"`. This seems to imply that you can declare a global variable inside a function?

```
# scoping-7.py
def h():
    global a
    a = 6
h()
print(a)
```

This also seems a bit counterintuitive from what we know of our past experience with programming languages. Nonetheless, this is how Python operates, and we must keep this in mind if we continue to use this language. These scoping rules also seem to impact what functions refer to.

```
# scoping-6.py
a = 5
def h():
    print(a)
h()
a = 6
h()
```

Notice here that this prints 5 and then 6. Not all languages have this behavior (like OCaml), and this becomes important when we talk about `closures`.

Consider the following and think about what this means when designing python programs:

```
# scoping-7.py
if True:
    b = 5
else:
    b = 6
print(b)
```

3.3.1 Nested Functions

In the same vein as scoping, we can define functions within other functions.

```
def outer_func(a):
    def inner_func(b):
        return a + b # can access 'a' in the inner scope
    # could not access 'b' here in outer scope
    return inner_func(4)

print(outer_func(5)) # prints 9
```

The inner function here can access all variables in the `outer_func` scope, but the reverse is not true. A common use case of this is when using higher order programming (a future chapter).

3.4 Object Oriented Programming

Python supports the Object Oriented Programming paradigm, which means we can use some of our java knowledge when working with classes. Let's first look at our syntax though:

```
# poop-1.py python object oriented programming
class Square:
    def __init__(self, size):
        self.size = size

    def area(self):
        s = self.size
        return s * s

s = Square(5)
print(s.area())
```

What looks new to you here? Perhaps some things to note:

- `class` keyword looks similar to java
- `__init__` must be a constructor of some sort
- `self` is a parameter and seems to be similar to java's `this`
- no new keyword when making a new object
- methods are defined inside the body of the class
- methods are called using the dot (.) syntax

- Much like functions, it seems like indentation matters here. We said earlier that the body of a function will always be indented. Additionally, we said there were no semicolons. Notice there are also no brackets either. This means Python must need some other way to figure out which lines of code are associated with each other. In this manner, new lines and indentation matter quite a lot in Python.

Noticing these things may raise some more questions. What happens if we want to inherit from a parent class? How exactly does `self` work? Is there a default `toString()` method?

To inherit from another class, we need to add a parameter to the class declaration:

```
1  # poop-2.py python object oriented programming
2  class Rectangle:
3      def __init__(self,width,height):
4          self.width = width
5          self.height = height
6
7      def area(self):
8          return self.width * self.height
9
10 class Square(Rectangle):
11     def __init__(self,size):
12         super().__init__(size,size)
13
14     def __str__(self):
15         return "Width: " + str(self.width) + "\tHeight: " + str(self.height)
16 r = Rectangle(4,5)
17 s = Square(5)
18 print(r.area())
19 print(s.area())
```

Notice there is a built-in `toString` method called `__str__`, and it can be overwritten. Notice that `super` is still a thing, and we can use it to call the parent's constructor. Lastly, notice that `self` is not explicitly passed in, but you can use it to refer to the current object's attributes.

3.5 Syntax

Here are just basic syntax things for the python language. At the end of the day, a for loop is a for loop, an array is an array. These are just simple syntax notes about these things. I will say that most languages will always have something of the following:

- Built-in data types/structures
- Control flow structures (for, if, while, jmp)
- I/O (print/read)
- variable assignment and manipulation
- functions
- comments

Knowing the syntax is important to learning a language, but it is also important to know the lingo a language uses. For example, Python does not have `booleans`, but rather they have `bools`. This may be pedantic and may sound pretentious to some (which is not entirely false), but consider:

```
1  # syntax.py
2  def f(x):
3      if type(x) == bool:
```

```
4         return 3
5     else:
6         return 4
```

Having line 3 be `if type(x) == boolean` would fail due to `boolean` not being something in the python language. Just keep this in mind as you learn more languages.

3.5.1 Data Types and Structures

Python has a few common data types built into the language. The typical suspects consist of `ints`, `floats`, `bools`, and `strs`. What may be shocking is that there is no `char` type in Python. Some examples:

```
1  1                    # int
2  1.2                  # float
3  1.5 * 2              # float
4  7//2                 # int
5  7/3                  # int
6  int(1.5)             # int
7  "hello"              # str
8  len("hi")            # int
9  "a" + "b"            # str
10 "na"*15 + "batman"   # str
11 True or False        # bool
12 True and False       # bool
13 not False            # bool
```

From previous languages, we are (hopefully) familiar with arrays and hashmaps. In python, we have lists and dictionaries. They also support tuples and sets(!).

```
1  a = [1,2,3]          # list
2  [1,"hi",True]        # list
3  sorted([3,2,1])      # list
4  a[0]                 # returns 1
5  a[-1]                # returns 3
6  a.append(4)          # list becomes [1,2,3,4]
7  a[1:]                # returns [2,3,4]
8  a[:2]                # returns [1,2]
9  a[1:3]               # returns [2,3]
10 a[1:3:-1]            # returns [3,2]
11 a[::-1]              # returns [4,3,2,1]
12 [1,2,3] + [4,5,6]    # returns [1,2,3,4,5,6]
13 a = {"key":"value",1:True} # dict
14 a["key"]             # returns "value"
15 a[1]                 # returns True
16 (1,2)                # tuple
17 (True,"Hello")       # tuple
18 (True,2.4,"hi")      # tuple
19 {1,2,3}              # set
20 {1,2,3,3}            # set that is just {1,2,3}
21 {"hi",1,False}       # set
```

3.5.2 Control Flow

It is always helpful to be able to control the order in which commands are executed, conditionally or unconditionally. In fact, it is one of the requirements for Turing completeness (something covered in a later chapter). The classic types of control statements are looping constructs and conditional execution (if, else).

Some looping constructs:

```
# conditional
if 3 > 4:
    print("False")
elif 4 > 5:
    print("Also False")
else:
    print("True")

# while loop
a = 0
while True:
    if a % 3 == 0:
        print("three")
    elif a % 2 == 1:
        continue
    elif a % 4 == 1:
        break

# for loop
for x in ["a","ab","abc"]:
    print(len(x))

for x in range(3):
    print(x)
```

Chapter 4

Higher Order Functions

> Higher Order Functions? More like lower suggestion inabilities
>
> Klef

4.1 Intro

We cover this topic in Python so the examples here will be mostly written in Python. A variation of this chapter written primarily in OCaml will follow this one.

4.2 Functions as we know them

Let us first define a function. A function is something that takes in input, or a argument and then returns a value. As programmers, we typically think of functions as a thing that takes in multiple input and then returns a value. Technically this is syntactic sugar[1] for the most part but that's a different chapter. The important part is that we have this process that has some sort of starting values, and then ends up with some other final value.

in the past, functions may have looks liked any of the following

```
\\ java
int area(int length, int width){
    return length * width;
}
/* C */
int max(int* arr, int arr_length){
    int max = arr[0]
    for(int i =1; i < arr_length; i++)
        if arr[I] > max
            max = arr[i];
    return max;
}

# Python
def str_len(str)
  return len(str)
```

[1] syntactic sugar just means that the syntax looks pretty or sweet like sugar

In these functions, our inputs were things like data structures, or 'primitives'. Ultimately, our inputs were some sort of data type supported by the language. Our return value is the same, could be a data structure, could be a 'primitive', but ultimately some data type that is supported by the language.

This should hopefully all be straightforward, a review and pretty familiar. The final note of this section is there are 3 (I would say 4) parts of a function. We have the function name, the arguments, and the body (and then I would include the return type or value as well). Again this shouldn't be new, just wanted this here so we are all on the same page.

4.3 Higher Order Programming

As we said, functions take in arguments that can be any data type supported by the language. A higher order programming language is one where functions themselves are considered a data type.

Let us consider the following C program:

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

int add1(int x){
  return x + 1;
}
int sub1(int x){
    return x - 1;
}

// return a function pointer
int* getfunc(){
  int (*funcs[2])(int) = {sub1, add1};
  return funcs[rand()%2];
}

// take in a function pointer
void apply(int f(int), int arg1){
  int ret = (*f)(arg1);
  printf("%d\n",ret);
}

int main(){
  int i;
  srand(time(NULL));
  for(i = 0; i < 5; i++){
    apply(getfunc(),3); //playing with pointers
  }
}
```

This program has one function that returns a function pointer, and one function that takes in a function pointer. The idea of this is the basis of allowing functions to be treated as data. For most languages we have the ability to bind variables to data.

```
int x = 3; // C, Java
y = 4 # python
// idea
// variable = data
```

If we consider what is going on in the machine (Maybe recall from 216), then we know that any piece of data is just 1s and 0s stored at some memory address. The variable name helps us know which memory address we are storing things (so we don't have to remember what we stored at address 0x012f or something). When we want to then refer to that data, we use

the memory address (variable name) and we retrieve that data. Why should a function be any different? We previously saw a pointer to a function being passed around, which just means the pointer to a list of procedures that are associated with the function. So in the case of higher order programming, we are just allowing functions to be passed in function data as arguments or be returned.

Thus we can say that a higher order function is one which takes in or returns another function. We can also avoid all these `void` pointers and casting and stuff in most functional languages.

4.4 Anonymous Functions

So we just said that we bind data to variables if we want to use them again. Sometimes though, we don't want to use them again, or we have no need to store a function for repeated use. So we have this idea of anonymous functions. It is anonymous because it has no (variable) name, which also means we cannot refer to it later. In python, we call these lambda functions. The syntax of a lambda function in python is

```
# add 1
lambda x: x + 1
(* add *)
lambda x,y: x + y
# general syntax
# lambda var1,var2,... varx: e
```

This is no different that just saying something like `2 + 3` instead of saying something like `x = 2 + 3`. This means that we can do the same thing by doing something like

```
2 + 3                  # expression by itself, no variable
x = 2 + 3              # expression then bound to a variable
lambda x: x + 1        # function by itself, no variable
add1 = lambda x: x + 1 # function bound to variable
```

Which means the following is just syntactic sugar.

```
def add1(x):
  return x + 1
# is syntactic sugar of
add1 = lambda x: x + 1
```

This is because the idea of functions as "first class data" is all based on this thing called lambda calculus, which is another chapter. But if we think about our mathematical definition of a function: it is something that takes in 1 input, and returns 1 output. So if each function should have 1 input, then what about functions that have multiple inputs? (`max(x,y)`?)

4.5 Partial Applications

Recall a section or something ago when we said that higher order functions can take in functions as arguments, and return functions as return values. Consider:

```
def plus(x,y):
  return x + y
```

We said earlier that functions have 1 input and 1 output. It seems here that this function has 2 inputs. We didn't lie, this is more syntactic sugar. Let us consider:

```
def plus(x,y):
  return x + y
def plus2(x):
  return lambda y: x + y
```

Here `plus2` is a function that takes in an `int` but then returns a function that itself takes in an `int` and returns an `int`. Which means we can actually define plus as

```
def plus():
  lambda x: lambda y: x + y
plus(2)(3)
```

If we can define functions like this then we can do things like

```
plus = lambda x: lambda y: x + y
add3 = plus(3)
add3(5) # returns 8
```

This is called a partial application of a function, or the process of currying. Not all functional languages support this unless the function is specifically defined as one which returns a function.

It is important to note here that you can only partially apply variables in the order used in the function declaration. That is a function like `add = lambda x: lambda y: x + y` can only partially apply the x variable: `add4 = add(4)`. This is because we are technically doing something like `add4 = lambda y: 4 + y`.

To be more clear:

```
def sub(x,y):
  return x - y
# same as let sub = lambda x : lambda y: x - y
def minus3(y):
  return sub(y,3)
# minus3 = fun y -> y - 3
```

So how does functional and currying supported languages know what the values of variables are? Or how are partially applied functions implemented? The answer lies with this idea of a closure.

4.6 Closures

A closure is a way to create/bind something called a context or environment. Consider the following:

```
def and4(w,x,y,z):
  return w and x and y and z
# and4 = lambda w: lambda x: lambda y: lambda z: w and x and y and z
def and3():
  return add4(True)
# and3 = lambda x: lambda y: lambda z: True x and y and z
def and2():
  return and3(True)
# and2 = lambda y: lambda z: True and True and y and z
```

How does the language or machine know that you want to bind say variable `w` to `true`? To be honest, there is no magic, we just store the function, and then a list of key-value pairs of variables to values. This list of key-value pairs is called an environment. A closure is typically just a tuple of the function and the environment. Visually, a closure might look like the following:

```
def sub(x,y):
  return x - y
def sub3(x):
  return sub(x,3)
# sub3 may look like
# (function: lambda x: lambda y: x - y, environment: [y:3])
#
```

It is important to note how a language deals evaluates a closure. In some languages, the binding of variable to values occurs when the closure is created, and in others it occurs when the closure is called. In Python, the variable lookup is done when the closure is called. Consider the following Python code snippet

```
x = 5
a = lambda y: x + y
print(a(5)) # prints 10
x = 3
print(a(5)) # prints 8
```

It is important to note, that when you have a local variable that is being bound, this will be bound independently of the global or higher scope variable. Consider

```
sub = lambda x: lambda y: x - y
x = 3 # this is a different x than in the line above
subfrom3 = sub(x)
print(subfrom3(5)) #prints -2 (3-5 = -2)
x = 5
print(subfrom3(5)) # still -2
```

4.6.1 Nested Functions

It may be easier to visualize what a closure is when using nested functions.

```
def outer_func(a):
    def inner_func(b):
        return a + b
    return inner_func
```

From a previous chapter you may remember this example with a slight modifcation. In this case we are returning the function `inner_func`, rather than a call to the function (`inner_func(4)`. In this case, we could visualize the closure as lines 2-3 along with whatever value `'a'` is.

```
c = outer_func(5) # c could be considered a variable that holds a closure
'''
If we want to visualize a closure, we could say that c is a variable that points to:
c = (def inner_func(b): return a + b, a = 5)
We can then call the closure say with
'''
d = c(5) # this will evaluate the closure substituting b with 5 and returning 10.
```

4.7 Common Higher Order Functions

Part of the reason why higher order functions (HOFs) are so useful is because it allows us to be modular with out program design, and separate functions from other processes. To see this, consider the following that we saw earlier:

```
def sub(x,y):
  return x - y
def div(x,y):
  return x / y
def mystery(x,y):
  return (x*2)+(y*3)
def sub3(y):
  return sub(y,3)
def div3(y):
  return div(y,3)
def double(y):
  return mystery(y,0)
```

Being able to make similarly structured functions into a generic helps makes things modular, which is important to building good programs and designing good software. We will see this with two very common HOFs: map and fold/reduce.

4.7.1 Map

Let us consider the following functions:

```
def add1(lst):
  ret = []
  for x in lst:
    ret.append(x + 1)
  return ret

def times2(x):
  ret = []
  for x in lst:
    ret.append(x*2)
  return ret

def isEven(x):
  ret = []
  for x in lst:
    ret.append(x%2 == 0)
  return ret
```

All of these functions aim to iterate through a list and modify each item. This is very common need and so instead of creating the above functions to do so, we may want to use this function called Map. Map will *map* the items from the input list (the domain) to a list of new item (co-domain). To take the above function and make it more generic, let us see that is the same across all of them:

```
def common(lst):
  ret = []
  for x in lst:
    ret.append(___)
  return ret
```

If we think about how we modify x, we will realize that we are just applying a function to x. Since it's the function that changes, we probably need to add it as a parameter. So adding this we should get

```
def common(lst,f):
  ret = []
  for x in lst:
    ret.append(f(x))
  return ret
```

Fun fact: map actually exists in Python(`map(lambda x: x + 1),[1,2,3])`. Either way, in OCaml and other languages without imperative looping structures, this is a common recursive function that is needed and can be used to modify each item of a list. Consider the code trace for adding 1 to each item in a `int list`.

```
def common(lst,f):
  ret = []
  for x in lst:
    ret.append(f(x))
  return ret

common([1,2,3],lambda x: x + 1)
'''
lst = [1,2,3]
f = lambda x: x + 1

starting at line 2 we set the initial value of ret
```

```
ret = []

then we iterate over the list of [1,2,3]
iteration 1:
  x = 1
  ret = []

  We then added an item to ret: ret.append(f(x))
  f(x) is the same as (lambda x: x + 1)(1)
  so ret.append(2) or since ret is [], [].append(2)
  so ret is now [2]

iteration 2:
  x = 2
  ret = [2]

  We then added an item to ret: ret.append(f(x))
  f(x) is the same as (lambda x: x + 1)(2)
  so ret.append(3) or since ret is [2], [2].append(3)
  so ret is now [2,3]

iteration 3:
  x = 3
  ret = [2,3]

  We then added an item to ret: ret.append(f(x))
  f(x) is the same as (lambda x: x + 1)(3)
  so ret.append(4) or since ret is [2,3], [2,3].append(4)
  so ret is now [2,3,4]

we now finish the loop and return ret: [2,3,4]
'''
```

4.7.2 Fold/Reduce

Depending on what language background you have, the following concept is called either fold or reduce.

Consider the following functions and what they have in common:

```
def sum(lst):
    final = 0
    for x in lst:
        final = final + x
    return final

def product(lst):
    final = 1
    for x in lst:
        final = final * x
    return final

def ands(lst):
    final = True
    for x in lst:
```

```
16          final = final and x
17      return final
18
19  def length(string):
20      final = 0
21      for x in string:
22          final = final + 1
23      return final
24
25  def concat(lst):
26      final = ""
27      for x in lst:
28          final = final + x
29      return final
```

These types of functions will go through a list and combine all the values in the list to a single value (it will reduce a bunch of things to one thing). Despite these looking a little more complex than the map examples, notice there is still much commonality between these examples.

Let us first consider what is exactly the same with all of these examples: a list (a string is just a list of characters), a final value, a looping construct that goes through the inputted list.

```
1  def common(lst):
2      final = _
3      for x in lst:
4          final = ___# update final value
5      return final
```

Now to make this generic structure more useful, we can first recognize that the initial value (where we first set `final` to a value) is dependent on the purpose of the function. In order to abstract this out, we can add it as a parameter:

```
1  def common(lst,initial):
2      final = initial
3      for x in lst:
4          final = ___# update final value
5      return final
```

Next we can consider that we need to update the final value. To do this, we need some sort of function that takes in two values: `final` and the value to combine with `final`. Notice that in every case except the penultimate one, this value to be added is an item in the list.

```
1  def common(lst,initial):
2      final = initial
3      for x in lst:
4          final = update_func(final,x) # what to do when doing length?
5      return final
```

Now `update_func` is not defined so we should add it as a parameter:

```
1  def common(lst,initial, update_func):
2      final = initial
3      for x in lst:
4          final = update_func(final,x) # what to do when doing length?
5      return final
```

Now if we wanted to write the above (sans `length` for now), we could do the following:

```
1  sum = common(lst,0,lambda x, y: x + y)
2  product = common(lst,1,lambda x, y: x * y)
3  ands = common(lst,True,lambda x, y: x and y)
4  concat = common(lst,"",lambda x, y: x + y)
```

Much more compact and notice that both `sum` and `concat` use the same function! This allows us to be more modular with our program design, as well as lead to less code duplication.

Now what about `length`? How do we use `common` to do this? There is no magic here, we can decide to just not use the parameter.

```
length = common(lst,0,lambda x, y: x + 1)
```

Now in python, this is called `reduce` and the order of the parameters are slightly different, but the concepts are the same:

```
sum = reduce(lambda x, y: x + y,lst,0)
product = reduce(lambda x, y: x * y,lst,1)
ands = reduce(lambda x, y: x and y,lst,True)
length = reduce(lambda x,y: x + 1,lst,0)
concat = reduce(lambda x, y: x + y,lst,"")
```

There are other common or useful HOF that exist (like `filter`), but this is a good introduction to HOFs and how to use them. As an exercise, try writing `map` in terms of `reduce`. For another exercise, look into `filter` and write that in terms of `reduce`.

4.8 Non List Map and Fold

Putting this here so it's written down. The above functions are typically done upon lists, but ultimately can be done upon any data structure. For example, `map` over a tree would go through each node in the tree and modify the value at each node. Similarly, a `reduce` over a tree wold visit every node in the tree and combine their values (eg. sum over a BST).

Chapter 5

Regular Expressions

```
[A-Z][a-z]+\d{4}
```

Chatbot+9000

Regular expressions is something that is programming language independent. Since I started teaching this course, we have changed what language we used when going over this topic. Thus, this chapter will start off language independent before going into the language dependent sections. While you can skip the sections of the languages we are not using, it may be helpful to see regular expressions are used in other languages.
Additionally regex comes from the theory related very closely to finite automata which is a later chapter so for now we will do just the basics and a surface level of the topic here.

5.1 Introduction

For those that do not know, programs live in RAM on the machine. Since RAM is wiped everytime you power down your machine[1], programs and program memory is designed to live short term. For long term storage, we need to use hard drives, since what is written to a hard drive is saved for a much longer period of time. When we write to the hard drive we typically do this by writing a file and saving it.

One defining feature of the UNIX family is the idea that everything is treated as a file, and for the most part, this works fine since everything that is stored, is stored as a file. We have various file types and file descriptors but ultimately whenever we need to save data for long-term storage, we save it to a file (which ultimately is just a segment of bytes in memory). This also means that whenever we need to load something from storage, we need to open up a file and read from it.

Opening a file and getting the data from it is the easy part. The hard part is to try and parse and make use of the data. Typically from a coding perspective, you open a file, read from it and then have a string of data. Being able to properly and efficiently parse this string is what loading typically is. Examples of this is reading a save file for a videogame, loading an image from a `.png` file, loading settings from a config file. Regardless reading and parsing strings is important, but can be hard to do.

Many would think that if you are trying to read certain data from a string, that methods that deal with strings is the solution. If I loaded a configuration file and wanted to know if I had to set a value to **`'true'`** then I would probably want to check if the string has a subtring of **`'true'`**. This is certainly one way to solve this problem, but it soon gets very inefficient with large amounts of data. The solution to some of our problems here is Regular Expressions, or more commonly known as RegEx.

5.2 Regular Expression as a Tool

At a basic level, a **Regular Expression** is a pattern that describes a set of strings. At a deeper level, a regular expression defines a regular language, a language which can be created from a finite state machine. More on regular languages in a bit.

[1] for the most part. There is always the Cold Boot Attack

A finite state machine will be covered in a later chapter as well. For now however, we can think of regex as a tool (or library) used to search (and extract!) text.

When we wish to define a pattern to describe a set of strings, there are a few things that we must consider.

- An Alphabet - An **Alphabet** is the set of symbols or characters allowed in the string. If our set of strings are for English words, we would have a different alphabet than one that describes a set of mandarin strings.
- Concatenation - Since most strings are longer than a single character, we will need some way to demonstrate the concatenation of single characters to create longer strings.
- Alternation - Being able to say one thing or another. We could say that any non-empty set is a union of 2 other sets. So I want to say that a string in set S is in either S_1 or S_2 where $\{S_1, S_2\}$ is a partition of S.
- Quantification - The thing about patterns is that repeat. So if I want to have a pattern, then I need to allow for repetition.
- Precedence - Ultimately not something that is the basis of regex, but is helpful making sure we know what exactly is being described.

We will talk more about all of this in a future chapter.

5.3 Regular Expression Creation

Now that we have an idea of what regular expressions are, let us see how we can create a regular expression. To reiterate, a regular expression describes a set of strings. This means we need some way to describe a set. In mathematics, there are universally accepted ways to describe a set. For example: $\{x \in \mathbb{N} | \exists y \in \mathbb{Z} \land y = 2x\}$. This is the very common **set builder** notation. The mathematical community has agreed that this is valid syntax to describe a set. Analogously, the CS community has agreed there should be a special syntax to describe a set of strings called regular expressions.

In my experience, most languages use the POSIX-EXT standard for the syntax of a regular expression. However different languages may have different support.

To begin, let's make our very first regular expression. I will surround a regular expression with the '" (backtick) characters. This is just my notation, so refer to the future sections to see how to write a regular expression in specific languages.

```
‘a‘
```

That's it. That's our very first pattern. What does it mean?
This is a very boring regular expression, it just describes a set with a single element of the "a" string: {"*a*"}.

What about something longer? Maybe my name?

```
‘cliff ‘
```

This is again, quite boring. It's just the small set of the string "cliff": {"*cliff*"}. What if we wanted to describe a set of greetings?

```
‘hello|hi|good morning|sup|salutations ‘
```

Now we are getting somewhere. We now have something that describes the set: {"hello", "hi", "good morning", "sup", "salutations"}. Additionally, we have just seen our first regex operator: the `or` operation, denoted with a pipebar (|)[2]. We can use this to describe the set of digits.

```
‘0|1|2|3|4|5|6|7|8|9‘
```

Now for the first challenge: describe a set of strings that look like "I am x years old", where x is 0 through 9, inclusive.

To approach this problem, we could brute force it:

[2] One thing to note about this and all other operators: if you wanted to describe a set that includes the pipebar ({"|"}), then you would have to escape this operator: `‘(true|false)\|\|(true|false)‘`

```
‘I am 0 years old|I am 1 years old|I am 2 years old|...‘
```

While this does work, notice there is a lot of repetition and this can get annoying should we want more than just 10 ages. This would be equivalent of describing a set of the first 100 naturals by just listing them out: $\{0, 1, 2, 3, 4, 5, ...98, 99\}$. This is cringe and so instead we rather use set builder notation: $\{x \in \mathbb{N} | x < 100\}$. In regular expressions case, we also have a shortcut to help us out here.

```
‘I am (0|1|2|3|4|5|6|7|8|9) years old‘
```

Notice that in our previous patterns when we used the `or` operator, the scope of the operator extended until the start or end of the pattern, or until another `or` operator. So something like `‘there’s one|two|three dogs‘` describes the set {"there's one", "two", "three dogs"}.

To get around this issue, we can put parenthesis to change the scope of the operator. So we could describe the set {"hi cliff", "bye cliff", "hi clyff", "bye clyff"} using the following regular expression:

```
‘(hi|bye) cl(i|y)ff‘
```

Now while this shortcut is helpful, it's still a little annoying. What if I wanted to include more numbers? I don't really want to do `‘0|1|2|3|4|5|...|99‘` because this is just the previous problem of repetition I just mentioned. Thus, let me introduce the next shortcut: quantification.

If I want to repeat a string multiple times, we can do this by adding a quantifier to a pattern (or sub-pattern). There is only 1 true quantifier, called the `Kleene operator`: *. This operator will repeat said pattern 0 or more times. Let's see what that means

```
‘(ha)*‘
```

This pattern describes the set: {"", "ha", "haha", "hahahha", "hahahaha", ...}. It is important to note that zero or more times includes the empty string. So if I wanted to describe any number, I can use the following pattern:

```
‘(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*‘
```

This regular expression describes a digit followed by 0 or more digits. This itself has a shortcut because this can also get long:

```
‘(0|1|2|3|4|5|6|7|8|9)+‘
```

The + operator is used to denote 1 or more repetitions. Other quantifiers include the following:

```
‘(0|1|2|3|4|5|6|7|8|9)?‘ - one or zero times
‘(0|1|2|3|4|5|6|7|8|9){x}‘ - x times
‘(0|1|2|3|4|5|6|7|8|9){x,}‘ - at least x times
‘(0|1|2|3|4|5|6|7|8|9){,y}‘ - at most y times
‘(0|1|2|3|4|5|6|7|8|9){x,y}‘ - between x and y times (inclusive)
```

Now these shortcuts are helpful, and cut down the length of the pattern, but imagine if we wanted to describe any lowercase string of size 3.

```
‘(a|b|c|d|...|x|y|z){3}‘
```

This would be a very long pattern leading to the last few shortcuts: bracket expressions. Bracket expressions allow us to describe ASCII ranges,or characters together,or even denote anything but a series of characters.

```
‘[a-z]{3}‘ - any ASCII character between a-z (inclusive) repeated thrice
‘[0-9]+‘ - any ASCII character between 0-9 (inclusive) repeated at least once
‘[aeiou]‘ - any vowel. This is equivalent to ‘(a|e|i|o|u)‘
‘[a-zA-Z]‘ - any upper or lowercase letter. Equivalent to ‘[a-z]|[A-Z]‘
‘[^abc]‘ - anything EXCEPT an "a","b", or "c"‘
```

Now brackets expressions can help us describe large ranges of characters, but what if we wanted to describe any character? Maybe something like any string of size 3? This can be solved with the dot character (‘.‘). The dot character to represent any character.

```
‘.{3}‘ - any sting of length 3
‘.at‘ - any string of length 3 that ends with ‘at‘.
```

It is important to note that the dot, the or, and the brackets only represent a single character. It is only when we add a quantifier, do we then repeat. To also be very clear, these all describe strings. ‘[0-9]‘ describes a string that represents a one digit number, but the set described consists of only strings.

5.4 Regular Expression Examples

Consider the following regular expressions and think about what types of strings they describe.

- ‘-?[0-9]+‘ - strings of integers
- ‘-?[0-9]+[.][0-9]+‘ - strings of floats with or without leading and trailing zeros
- ‘[A-Z][a-z]*‘ - strings that begin with a capital followed by zero or more lowercase characters
- ‘[AEIOUaeiou]{5,10}‘ - strings consisting of only vowels of length 5 - 10
- ‘I am [1-9]?[0-9] years old‘ - I am x years old where $0 \leq x \leq 99$
- ‘I am ([2-4][0-9]|50) years old‘ - I am x years old where $20 \leq x \leq 50$

5.5 Regular Expression Matching

Now that we know how to describe our set, we need some way to check if a string (or part of a string) is in the set. This is where we get into the usage of regular expressions for practical use. So we will now start talking about how languages use regular expressions to solve problems.

If a string, *s*, is in the set described by the regular expression *r*, then we say that *r* **accepts** s. If *s* is not in the set, then we say that *r* rejects *s*. Checking if a regular expression accepts or rejects a string can be done in O(n) time where *n* is the length of the string. We will talk more about how to implement this later, but the great thing is because it’s linear, we can actually check if any part of a string is in the set described by the regular expression.

Since we can check if any part of a string is in the set, we may need to denote we want an **exact match** over a **partial match**. An exact match is one where the entire string matches the regular expression. Most languages I have worked with perform a partial match, although I am aware that other languages default to an exact match.

To demonstrate this, let’s assume that there is some function called `match(str,re)` that has two parameters, a string `str` and a regular expression `re`. It returns `true` if `re` accepts `str` and `false` otherwise.

```
If using an exact match:
match("unknowing",‘know‘) returns false since "unknowing" is not in {"know"}
If using a partial match:
match("unknowing",‘know‘) returns true since "know" can be found in "knowing"
```

If the language’s default is a partial match, we have two other symbols to help us achieve something similar to an exact match.

```
If performing a partial match
‘^ch‘ - any string that begins with ‘ch‘
‘at$‘ - any string that ends with ‘at‘
```

The carot symbol (^) is something we already saw as something akin to a **not** operator in a bracket expression. When used outside of a bracket environment, it is used to denote the start of the string.

```
If performing a partial match:
‘^(this|that)‘ - any string that begins with ‘this or that‘
‘^this|that‘ - any string that begins with ‘this‘ or contains ‘that‘
```

The dollar sign ($) is the symbol for denoting the end of the string.

```
If performing a partial match:
‘(this|that)$‘ - any string that ends with ‘this or that‘
‘this|that$‘ - any string that ends with ‘that‘ or contains ‘this‘
```

Putting these two together allow us to obtain exact match functionality.

```
Regardless of performing a partial or exact match:
‘^(this|that)$‘ - either the string "this" or the string "that"
```

Now this process is very helpful for checking if a string matches a certain pattern, but this is only so useful. Sometimes it is useful to actually extract parts of a string based on a pattern.

5.6 Regular expression Grouping

While we can check for acceptance of a string, regex is more useful to actually extract data from a string. For example, you have a file and it has a 1000 lines of phone numbers (eg. `"John Smith:123-4567890"`). Your job is to sum up all the area codes.

To do this, we need some way to mark certain parts of the strings to be stored somewhere. Most implementations of regular expressions just use the parenthesis to do this. So if I wanted to mark the area code, I could do the following:

```
if doing a partial match:
‘([0-9]{3})-[0-9]{7}‘ - put () around the area code part: ([0-9]{3})
If doing an exact match:
‘[A-Z][a-z]+ [A-Z][a-z]+:([0-9]{3})-[0-9]{7} - this makes assumptions about what a valid name is
```

Now that we marked what we wanted to extract, we need some way of obtaining the extracted parts. This is language dependent, but the convention of what order the groups are is not. The previous example is simple in the sense there is only 1 set of parenthesis. What happens if we use a more complicated pattern?

```
‘Cl(i|y)ff’s phone number is (([0-9]{3})-([0-9]{7}))‘
```

Here we have marked four (4) parts (groups). The first group if apart of the `‘Cl(i|y)ff‘` segment. This was more to allow an more alternate spelling of my name, but regex will still extract either the ’i’ or ’y’ and store it in group 1.

Continuing from that, we read the regex left to right, and whenever we see a left (open) parenthesis, that is the next group. Thus, group 2 is the entire phone number, group 3 is the area code, and group 4 is the rest of the phone number.

5.7 Regular Expressions in Programming Languages

5.7.1 POSIX, POSIX-EXT, PERL Syntax

If there is one thing that is constant in life, there is always someone who thinks they can do something better that someone else. The idea of Regex and what is included in the syntax has a few different standards. You can read about them here. In fact, some of the things we introduced (like the {count} syntax) are not found in basic regular expressions. Most common shortcuts stem from Perl or POSIX-EXT standard. Some languages have their own rules about what syntatical things are included or not. Just keep this in mind when applying regex to a language.

Tl;Dr: Read the documentation of the language when writing a regex and using shortcuts/syntactic sugar.

5.7.2 Regular Expression In OCaml

Let’s start by making our first pattern. To begin, we will need to include the `re` library. In `utop` you need to run `#require "re"`. Once we have done that, we can start using regular expression methods.

```
#require "re" (* only in utop *)
let comp_re = Re.compile (Re.Posix.re "I am [0-9]+ years old") in
let did_matched = Re.execp comp_re "I am 23 years old" in
if did_matched then
  print_string "Successfully matched"
else
  print_string "Did not match";;
```

The `re` module has many useful functions that expect a regular expression type. The `re` library has many ways to convert a string to a regular expression. Some languages use a different set of operations and syntax than others. POSIX-Extended standard is common as is Perl. The `Re.Posix.re`, is one such conversion function.

Once you have a regular expression type, you will need to "compile" it, to be efficient to use for searching and extracting text. The `Re.execp` function will return `true` or `false` if the regex accepts the string. The default behavior is a partial match. If you want to extract data, you will need to use the `Re.exec` or `Re.exec_opt` functions.

```
#require "re" (* only in utop *)
let comp_re = Re.compile (Re.Posix.re "I am ([0-9]+) years old") in
let matched = Re.exec comp_re "I am 23 years old" in
print_string ("Age: " ^ (Re.Group.get matched 1))
```

If you wanted to not write a string for a regex, the `re` library has a few types and functions you can use to create a regular expression type as well. The above could also have been written as:

```
#require "re" (* only in utop *)
let comp_re = Re.compile (Re.seq [Re.str "I am "; Re.group (Re.rep1 Re.digit); Re.str "
    years old"]) in
let matched = Re.exec comp_re "I am 23 years old" in
print_string ("Age: " ^ (Re.Group.get matched 1))
```

You can read more at `https://ocaml.org/p/re/latest/doc/Re/index.html`.

5.7.3 Regular Expression In Python

Let's start out on making our first pattern. To begin, we need to include the regular expression module which we can do with the `import re` command. Once we have done that, we can start using regular expression methods.

```
# regex.py
m = re.compile("I am \d+ years old")
matched = m.match("I am 23 years old")
if matched:
  print("Successfully matched")
else:
  print("Did not match")
```

The return value of `m.match()` is a `matched_data` type which has information regarding what you are trying to extract.

```
# regex.py
import re
m = re.compile("I am (\d+) years old")
matched = m.match("I am 23 years old")
if matched:
  print("I am " + m.group(1)))
else:
  print("Did not match")
```

Calling `match()` on a compiled regular expression in Python will check if the string matches from the beginning of the input string. Thus, `match()` performs a semi-partial match. Other methods exist that do a partial match anywhere in the

string, or an exact match (`search()`, `fullmatch()`).

You can actually test and see what is accepted and captured using this fun online tool called `https://pythex.org/`. That or you can play around with the code segments found in this chapter and see what happens.

You also don't need to compile the regex if you aren't going to use it over and over. (It is more efficient to compile if it will be repetitively used a lot).

```
# regex.py
import re
matched = re.match(r'I am (\d+) years old','I am 23 years old")
if matched:
  print("I am " + matched.group(1)))
else:
  print("Did not match")
```

5.7.4 Regular Expression In Ruby

Let's start out on making our first pattern. Much like `Strings` denoted with single or double quotes, and arrays with the `[]` symbols, and hashes using `{}`, patterns are surrounded by the forward-slash: `/`.

```
# regex.rb
p = /pattern/
puts p.class #Regexp
```

This pattern is the string literal **`'pattern'`**. That is, this pattern describes the set of strings that contain the substring **`'pattern'`**. Not a very fun pattern, but a pattern nonetheless.

But Great! We have a pattern. Now how do we use it? There are 2 main ways that we can match this pattern that I know of, one of which I like and the other I do not. We will go over the latter because it is easier version and my reason for not liking it it petty.

```
# regex-1.rb
p = /pattern/
if p =~ "pattern" then
  puts "Matched"
else
  puts "Not matched"
end
```

If everything goes as planned, running this file will have `"Matched"` printed. Why does this happen? `=~` is a method associated with Regexp objects which take in a string and returns an integer or nil depending on if the pattern can be found *anywhere* in the string. Thus Ruby performs partial matching be default. If the pattern is found, it will return the index of the first character of the first instance of the pattern.

Since using the `=~` method will search the entire string for a match, we will have to use your start and end of string specifiers.

```
# regular_expressions.rb
if /^where/ =~ "anywhere in the string" then
  puts "matched at the begining"
end
if /where$/ =~ "anywhere in the string" then
  puts "matched at the end"
end
if /where/ =~ "anywhere in the string" then
  puts "matched somewhere in the string"
end
```

Here the only thing printed is `Matched somewhere in the string`.

Now that you can check to see if a string matches a pattern, we can move onto grouping. Ruby uses the global variables `$1, $2, ...,$x` to refer to the first, second, ..., xth capture group.

```
# capture.rb
pattern = /([0-9]{3})-[0-9]{7}/
if pattern =~ "111-1111111" then
    puts $1 #111
end

pattern = /(([0-9])[0-9]{2}))-[0-9]{7}/
if pattern =~ "123-4567890" then
    puts $1 #123
    puts $2 #1
end
```

One other important thing to note is that whenever the =~ method is called, then these top level variables $1, $2, etc, are all reset. This is the main reson as to why I do not like this method of matching patterns despite how easy it is.

The way that I prefer to match patterns is by using the `match` method. This instead returns an array of all matched groups (if any) which means I can store the results to a variable and refer to them later and not worry about data being wiped. That's it, the only reason why I do not like the previously described method.

You can actually test and see what is accepted and captured using this fun online tool called `http://rubular.com`. That or you can play around with the following code segments and see what happens.

```
# capture-2.rb
pattern = /[A-Z][A-Z]*/
strs = ["a","A","abcD","ABDC"]
for test_string in strs
  if pattern =~ test_string
    puts "matched"
  else
    puts "not matched"
  end
end

#what if the pattern and test strings are as follows:
pattern = /a[A-Za-z]?/
strs = ["a","abd","bad"]

pattern = /^a[A-Za-z]?$/
strs = ["a","abd","bad"]

pattern = /a*b*c*d*/ # how is this different from /[a-d]/ ?
strs = ["abcd", "bad", "cad", "aaaaaacd", "bbbdddd"]

pattern = /^(..)$/
strs = ["even","odd","four","three","five"]

# an even number of vowels
pattern = /^([^aeiou]*[aeiou][^aeiou]*[aeiou][^aeiou])*$/
```

Chapter 6

OCaml

> We will do so much Ocaml, you could call it Ocamlot
>
> Cleff

This is a programming language chapter so it has two (2) main things: talk about some properties that the OCaml programming language has and the syntax the language has. If you want to code along, all you need is a working version of OCaml and a text editor. You can check to see if you have OCaml installed by running `ocaml -version`. At the time of writing, I am using Ocaml 4.14.0. `ocaml` is repl which you can use to play around with OCaml but please use `utop`. It's a wrapper for `ocaml` and is much easier to use.

6.1 Introduction

OCaml will probably look (and act) unlike any other language you have come across in the CS department up to this point. This is due to one key difference: OCaml is a declarative programming language, opposed to an imperative language. We will talk about this all in the next section, but for now let's just write our very first program.

```
1 (* helloWorld.ml *)
2 print_string "hello world"
```

Despite this being very simple, we observed five (5) things.

- Comments are surrounded by `(* ... *)`
- no semicolons to denote end of statement*
- `print_string` is used to print things out to stdout
- Parenthesis are optional when calling functions*
- OCaml file name conventions are `camlCase.ml`
- Strings exists in the language (most languages do, but some do not)

Now you may have noticed the $*$ symbol over point 2 and 4. That is because these observations are actually False. Or at least, not entirely true. For now though, let's just roll with it. In order to have the above code run, you can run

```
ocamlc helloWorld.ml
./a.out
```

Congrats, you have just made your first program in OCaml! There are some things to note however:

- OCaml is a compiled Language

- `ocamlc` is the ocaml compiler (some compilers like to take the name of the language and add 'c' to the end: javac, ocamlc, rustc).
- If you run an `ls` you will notice that along with the executable `a.out`, two other files were generated as well, `helloWorld.cmo` and `helloWorld.cmi`. The `.cmo` is the object file and can be thought of as analogous to the `.o` file generated by gcc. The `cmi` file is the interface file and can be thought of as analogous a compiled down `.h` file in c.
- `ocamlc` is wrapped in a nice program called dune. dune will allow you to compile, run, and test your OCaml programs without much overhead. We use dune to help manage your projects.

6.2 Type System

As mentioned in the python/ruby chapters, a type system dictates the how data is treated, along with what you can do with certain pieces of data. To recap, let's first talk about type checking.
Type checking is the process of determining what a piece of data's type is. Should the bitstring `0100000101000001` be treated as a string and have the value of "AA" or should it be treated as an int with the value of 16705? Additionally, *when* should this checking occur? At compile time? Runtime? Different type systems have different answers to these questions. Let's first start with then when.

Static typing is something you are familiar with from Java and C: type checking is done at compile time. In Python or Ruby where there is no compiler[1], type checking is done at runtime. This is called dynamic typing. Static and Dynamic typing are contrasting and only dictate *when* type checking should occur. To talk about how types are checked, then we need to talk about other type systems.

Explicit typing is something you are familiar with in Java and C: the programmer must explicitly state the type at creation of the variable.

```
int i = 3;
String s = "hello';
```

The above are examples of explicit typing. This is very important in C because you can do whatever you want with whatever data for the most part. Explicitly typed languages are almost always associated with manifest typing (and sometimes these terms are used interchangeably). With manifest typing, types are associated with variables, not values. Thus, it is the variable `i` that has type `int`, and not the bitstring `00000011`.

In contrast to this is the idea of implicit typing (sometimes called latent typing), where the programmer does not need to put down the type of the variable. In this case, type inference occurs, where the language will infer the types of the data (we see this in OCaml) and we saw this in Python. In this case, types are associated with values and not variables.

It is important to note, that explicit and implicit type systems are independent of static and dynamic type systems. They do not imply each other. OCaml is a static and implicitly typed language. Python is a dynamic and implicitly typed language. C is a static and explicitly typed language.

6.3 Functional Programming

According to Wikipedia, "functional programming is a programming paradigm where programs are constructed by applying and composing functions". This probably means nothing to you, so let's make our own definition. First let's define a few words, or rather one in particular: paradigm. Depending on the field, it has many different definitions. I don't want to take the lingustic's definitions, despite that is the one used here. I want to take the more general one: a set of thoughts and concepts related to a topic. With this defintion, I will say this: **Functional programming is a way of programming that focuses on creating functions rather than listing out steps to solve a problem.**. I'm sure that many people will disagree and dislike this definition but oh well.

[1] python does do some compile like things tho

6.3.1 Declarative Languages

Functional languages are typically described as declarative. This means that values are declared and the focus is declaring **what** a solution is, rather than describing **how** a solution is reached. To see this let's do a real quick comparison in English for a process that finds even numbers in a list

Imperative
+ make an empty list called results
+ Look at each item in the list
+ Divide the item by 2 and look at the remainder
+ if the remainder is 0, add the value to the results list
+ return the results list after you looked at all list items

Declariative
+ Take all the values that are even divisible by 2 from the list
+ Return those values

Notice that the imperative instructions tell you *how* to do something and does so in steps. The declarative instructions tells you what you are looking for and assumes you can just figure out how to do it. If we translate the imperative code to Python we get something like

```
1 #imperative.py
2 def evens(arr):
3     results = []
4     for x in arr:
5         remainder = item % 2
6         if remainder == 0:
7             results.push(item)
8     return ret
```

Now we haven't learned enough (or really any) OCaml at this point to write a solution to this yet[2] , but let's look at a declarative python example:

```
1 #declarative.py
2 results = [x for x in arr if x % 2 == 0]
```

Here, we don't tell python *how* to solve the problem, we tell it what we want and python figures out the rest.

6.3.2 Side effects and Immutability

One thing that functional programming aims to do is to minimize this idea of a side affect. Consider the following Python Code:

```
1 # side_effects.py
2 count = 0
3 def f(node):
4   global count
5   node.data = count
6   count+=1
7   return count
```

Functional programming wants to treat functions as, well, a function. This means that something like the following should be true:

`f(x) + f(x) + f(x) = 3 * f(x)`

However, if we run the code above, then `f(x) + f(x) + f(x) = 1 + 2 + 3` and `3 * f(x) = 3 * 1`. This unpredictability is called a side effect (The true definition of a side effect is when non local variables get modified). When side effects occur, it becomes harder to reason and predict the behaviour of code (which means more bugs!). To combat this, OCaml makes all variables immutable to help maintain **referential transparency**. Referential transparency is the ability to

[2] If you wanted a solution here is one:
`let rec even lst = match lst with []-> []|h::t -> if h mod 2 == 0 then h::(even t) else (even t) in even lst`

replace an expression or a function with it's value and still obtain the same output. To simplify, OCaml wants to minimize the amount of outside contact your code has to make everything self contained. The more you rely on outside information or context, the more complicated your code becomes. Now we need to address the question, "What is an expression or function's value?"

6.3.3 Expressions and Values

In functional languages, one of the core ideas is ability to treat functions as data. Which means much like in python, we can pass functions in as arguments, or use them as return values to other functions. But we are getting a little ahead of ourselves. Let's first see what data is in OCaml.

In OCaml, we say that almost everything is an expression. [3]Expressions are things like `4 + 3` or `2.3 < 1.5`. We say that expressions evaluate to values. A value is something like 7 or `false`. All values are expressions in and of themselves, but not all expressions are values. Like a square and rectangle situation. All expressions also have a (data) type. The expression 3+4 evaluates to 7 so we say that both expressions have type `int`. For the purpose of these notes I will use e to represent an expression, t for type, and the structure $e : t$ to say that the expression e evaluates to type t. Consider the following:

```
(* expressions.ml *)
true: bool (* is a value, has type bool *)
3 * 4: int (* is an expression, has type int *)
"hello" ^ "world": string (* is an expression of type string *)
5.4: float (* a value of type float *)
```

Now, we said that almost everything is an expression but we do have one thing that I would not consider an expression: the binding of expressions to variables.[4] This can get confusing because there are these things called `let` *`bindings`* and `let` *`expressions`*. We will talk about the former first.

A `let binding` just binds an expression to a variable. We cannot use it where we typically expect an expression. Here is an example:

```
(* letBinding.ml *)
let x = 3 + 4
(* syntax *)
(* let variable = e*)
```

It is important to note that OCaml uses static and latent typing. Also recall that variables in OCaml are immutable. When we run the above code in a repl like *utop* we are actually setting a top level variable which can be used to refer in other places. We ultimately want to try and avoid this to maintain more strict referential transparency so we have this expression called a `let expression`.

A `let expression` is like setting a local variable to be used in another expression.

```
(* letExpression.ml *)
let x = 3 + 4 in x + 1
(* syntax *)
(* let variable = e1 in e2 *)
```

In this case, we add the `in` keyword and follow up with another expression. In this case, this is an expression so it does have a value it will evaluate to and also has a type. It's type is dependent on what the second expression's type is.

```
(let variable = e1:t1 in e2:t2):t2
```

We can of course nest these, and since data is immutable in OCaml, variables are overshadowed.

```
(* scoping.ml *)
let x = 3 in let y = 4 in x + y (* 7 *)
let x = 3 in let x = 4 in x (* 4 *)
let x = 3 in let z = 4 + x in let x = 1 in x + z (* 8 *)
(* implicit parenthesis *)
let x = 3 in (let z = 4 + x in (let x = 1 in x + z))
```

[3]Technically, everything is an expression due to specifications of OCaml, but in the broad scheme of things, I don't like this.

[4]Technically it is an expression (by ocaml specs), but is not the same as the other expressions we talk about;Languages like Scheme or Racket would not consider this to be an expression

Here, whenever we look consider a variable's value, it is always the closest preceding binding. Also, just to reiterate, these are expressions so something like the following is also possible:

```
let x = if true then false else true in let y = 3 + 4 in let z = if true then 2 else 6 in if
    x then y else z
(* implicit parenthesis *)
let x = (if true then false else true) in let y = (3 + 4) in let z = (if true then 2 else 6)
    in (if x then y else z)
```

Now that we have an an idea of what an expression is and how to determine some basic values and types, we can build larger expressions. I think of this as analogous as taking statement variables p and p and then building larger statements like $p \vee q$. And since we know how to evaluate basic expressions and find out their types and values, it's like knowing the truth values of p and q and being able to then conclude the truth value of $p \vee q$.

6.3.4 The if Expressions

Let us consider the very basic `if expression`. Now the `if expression` is an expression which means it has a value and type. But first let us consider it's syntax.

`(`**`if`** `e1:bool` **`then`** `e2:t e3:t):t`

What does this mean? As stated before, I will be using e to represent expressions and t for types, with $e : t$ meaing that e has type t. So in this case, the `if expression` has an expression $e1$ which must evaluate to a `bool`, and two other expressions $e2$ and $e3$ which must **both** evaluate to the same type t. The `if expression` as a whole then has that same type t. A little weird, let's see an example.

```
if true then 3 else 4
```

Here `true` is an expression of type `bool` and both 3 and 4 are expressions of type `int` which means this expression as a whole has type `int`. Now because we can substitute any valid expression for $e1$, $e2$, $e3$ as long as their types follow the above rules all the following are valid expressions:

```
if true then 3 else 4
if true then false else true
if 3 < 4 then 5 + 6 else 7 + 8
if (if true then false else true) then (if false then 3 else 4) else (if true then 5 else 6)
```

Unlike Python or C, the only things that evaluate to `bools` are `true` and `false` or expressions that evaluate to `true` and `false`. So `if 3 then 4 else 5` would be invalid. This idea of substituting any expression with the expected type can be used for any expression that has 'subexpressions'. So the following are valid with `let bindings` and `let expressions`:

```
let x = if true then false else true
let y = 3 + 4 - 10 in if true then y else y + 10
```

6.3.5 Functions as Expressions

We stated earlier that functional programming is one where we want to be able to treat functions as data. We have actually kinda saw this before. Consider the following:

```
x = 3
print(x)

def x():
    3
prints(x())
```

There is not much difference here as what is printed out or how we use the name `x`. In OCaml, I consider variables to be functions with no parameters that return a value very much like our `x` method above. This is because at the end of the day, if we recall out C and 216 days, a variable is just a way to refer to some specific memory address that holds data. That data could be a value, could be code. But an actual function definition looks like this:

```
(* functions.ml *)
let area l w = l * w
(* or to use a let expression where we call the function *)
let area l w = l * w in area 2 3
(* syntax *)
(* (let name e1:t1 e2:t2 ... ex:tx = e:ty):t1 -> t2 -> tx -> ty *)
```

Like `let bindings` a function definition by itself is not an expression. You will need the `in` keyword if you want it to be an expression. The type of a function is represented as a list of types that looks like `t1 -> t2 -> ... -> tx -> ty`, For example `let area l w = l * w in area` has type `int -> int -> int`. The last type in this list is always the return type, where the preceding types are the types for input.

The fun part is that since we know functions are expressions, and functions can take in expressions as input, then we can have functions that take in other functions.

```
(* functional1.ml *)
let area l w = l * w (* int -> int -> int *)
let apply f x y = f x y (* ('a -> 'b -> 'c) -> 'a -> 'b -> 'c *)
apply area 2 3
(* optional parenthesis *)
(* apply (area) (2) (3) *)
```

Now we have some new things to talk about here. Namely, what is `'a, 'b, 'c` and why is `area`'s type `int -> int -> int` and not something like `float -> float -> float`?

6.3.6 Type Inference

Let us consider the `area` function: `let area l w = l * w`. OCaml knows that this is a function with type `int -> int -> int`. Which means we cannot do something like `area 2.3 4.5`. Why is this and how does this work?

Type inference is a way for a programming language to determine the type of a variable or value. In some languages it's real easy because you explicitly declare types: `int x = 3;`. In OCaml, variables types are determined by the operations or syntax of the expression. So just like you can only use things like && and || on `bool`s, we can only use things like `+,-,*,/` on `int`s. If you wanted to do operations on `float`s then you need to use different operators. See the following:

```
(* operations.ml *)
2 + 3 (* int and int *)
1.3 +. 4.3 (* float and float *)
"hello" ^ " world" (* string and string *)
true || false (* bool and bool *)
int_of_char 'a' (* char input, int output *)
2 + 3.0 (* error *)
3 ^ 4 (* error *)
```

Some operators however, work on many different types. One such example is the > (greater than) operator. This operator along with `<, >=, <=, =` all can take in any two inputs as long as those two inputs are the same type. The output will always be of type `bool`.

```
(* compare.ml *)
2 > 4 (* false *)
"hello" <= "world" (* true *)
true = false (* false *)
```

One thing to note is that we use = for testing equality since we bind variables with `let ... = e`. So we can say something like `let x = 2 = 3` and OCaml knows that x is the varaible and anything after the first = sign is the expression. Anyway, this is important because then what type is inferred from a function like

```
(* typeInference0.ml *)
let compare x y = x > y in compare
```

Here we have no idea what type x and y have to be. In this case, OCaml uses a special type notation. The type of `compare` is `'a -> 'a -> bool`. That is, we have two inputs which must both be the same type, and we know the result will be of type `bool`. If we are given something even stranger like:

```
(* typeInference1.ml *)
let f x y = 3
(* this is equivalent to something like
def f(x,y)
    3
end
*)
```

OCaml will give this function type `'a -> 'b -> int`. We are returning an `int` but the input types could be anything. Since the input types don't even need to be the same type here, we give them different symbols. So let's go back to our `apply` function and break it's type down again.

```
(* typeInference2.ml *)
let apply f x y = f x y in apply (* ('a -> 'b -> 'c) -> 'a -> 'b -> 'c *)
```

First let's list out the parameter names: `f, x, y`. The first parameter is a function which has two inputs of unknown type. We also don't know if the two inputs have to be the same or different. At this moment we know that the function's type is `'a -> 'b -> ?`. Looking at the function we are given no information about what the return type of `f` is so we give it yet a another symbol. Thus we can say that the type of `f` is `'a -> 'b -> 'c`. Now we know that `f` is being called on parameters `x` and `y` so we know that `x` must be the type of `f`'s first argument so we can give `x` type `'a`. We then know that `y` is being used as `f`'s second argument so it should be of type `'b`. So at this point we know the type of `apply` is `('a -> 'b -> 'c) -> 'a -> 'b -> ?` (we put any function's type in parenthesis to show it's a function). Lastly we know that the returned value of `apply` is whatever `f x y` returns. Since we know that `f` returns some value of type `'c`, we can say `apply`'s return type is also `'c`. Thus the entire type of `apply` is `('a -> 'b -> 'c) -> 'a -> b -> 'c`

6.4 Ocaml Pattern Matching

The next feature that OCaml allows us to have is the ability to pattern match. Pattern matching is, if you squint, very closely related to a `switch` statement. While I could show you pattern matching with what we know, I find it easier to demonstrate once we know OCaml's built in data structure: `lists`.

6.4.1 Lists

In other languages you may used to having these data structures called Arrays. In OCaml we don't have arrays, we what we have instead is `lists`. Let's first see the syntax:

```
(* list.ml*)
[1;2;3;4] (* type: int list *)
(* syntax *)
(* [e1:t; e2:t; ... ex:t]*)
```

Looks a little like an Array but instead of being comma delimited, it is semi-colon delimited. Looking at the syntax we can also conclude that the `lists` must be homogeneous. Additionally, we can see that we do not need to put values, but rather we can put expressions. Here are some examples of other lists

```
(* list1.ml*)
[1;2;3;4] (* int list *)
[2.3;1.0] (* float list *)
["hello", "World"] (* string list *)
[2 + 3; 4-4; 7 * 9] (* int list *)
let f1 x = x + 1 in let f2 y = x + 1 in let f3 z = z + 1 in [f1;f2;f3] (* (int -> int) list
    *)
```

The last one is a list of `int -> int` functions, wild huh? Now it is important to note that `lists` are implemented as a linked list under the hood which means they are recursive.

6.4.2 Recursion

Now you may have noticed that I have not talked about `for, while, do while` or any other type of looping structure. That is because it does not exist in OCaml. We have something better: recursion. In OCaml, data structures are recursive and to do looping, we need to make recursive functions. We will talk about this is a bit. We first need to talk about lists. Now since `lists` are recursive, we need to consider how to define a `list`. If you recall from 132 your linked list data structure, you should recall that a linked list in Java is a 'node' which contains a piece of data and then points to another list or null. In OCaml, we don't use these words, but they can be used analogously. In OCaml we don't point to Null, but instead we point to an empty `list` which we call `Nil`. We then have an element which points to the rest of the list, which we use what we call the `Cons` operator.

```
[] (* Nil, the empty list *)
1 :: [] (* 1 cons Nil, add 1 to the empty list. Evaluates to [1] *)
1 :: 2 :: [] (* 1 cons 2 cons NIL. Evaluates to [1;2] *)
1 :: [2] (* 1 cons list of 2. Evaluates to [1;2] *)
(* syntax *)
(* e1:t :: e2:t list *)
```

Notice that the syntax shows that we are using expressions which means we have some wierd expressions that represent `lists`. Also note that the `cons` operator's left hand operator is of type `t` and the right hand operator must be of type `t list`.

```
[2 + 3; 4 - 5] (* int list. Evaluates to [5;-1] *)
[if true then false else true; false;] (* bool list. Evaluates to [false;false] *)
[[1;2;3];[4;5;6]] (* int list list. Is a value *)
[print_string "hello"; print_string "world"] (* Unit List. Don't worry about Unit now, but
    notice that "worldhello" is printed. *)
```

Notice that when we put expressions into lists, they are evaluated and not stored as expressions, but also evaluated from right to left order (see the last example).

6.4.3 Pattern Matching

So now that we have an idea of what a `list` is and how to construct one, now we have to learn how to deconstruct a `list`. Pattern matching is the way to deconstruct any data structure in OCaml and is a language feature not found in all languages. In order to pattern match, we need to learn a new expression: the `match` expression.

```
let x = 5
match x with
    0 -> 0
    |1 -> 1
    |3 -> 2
    |5 -> 3
    |_ -> 4
(* Syntax *)
(* (match e1:t1 with
     pattern1 -> e2:t2
    |pattern2 -> e3:t2
    | ...   ):t2
```

A `match` expression takes in an expression/value and then checks to see if it has the same structure as any of the cases. If it matches with a case, it will then preform the expression linked to the case. The last line is an underscore, which is used as a wildcard (match with anything else). Here is an analogous switch statement:

```
switch(5){
    case(0): return 0;
    case(1): return 1;
    case(3): return 2;
```

```
    case(5): return 3;
    default: return 4;
}
```

I don't want you to think of them as the same though, and pattern matching can do a lot more than a switch statement so just use this vaguely related but not the same. However like a switch statement, a match statement will check until the first pattern that satisfies the requirements and then not look at any of the other patterns. Additionally, notice the match expression is an expression which means it can be evaluated to a value and has a type. It's type is whatever each case returns, and so we need each branch to have the same return type. The next thing to discuss is the idea of a *pattern*. A pattern is not like a regular expression pattern, but it matches with how a piece of data could be represented. Consider all the ways we can represent a list of 2 items. We can use each of these in a math expression and they mean the same thing.

```
1 match [1;2] with
2     a::b::[] -> 0
3     |a::[b] -> 1
4     |[a;b] -> 2
```

In the above example, the expression evaluates to 0, but all of those patterns mean the same thing. Here is an example of pattern matching on a list where we return the length or 4 if longer than 3 elements.

```
1 (*let lst = ... some list *)
2 match lst with
3     [] -> 0
4     |[a] -> 1
5     |a::b::[] -> 2
6     |a::[b;c] -> 3
7     |h::t -> 4
```

In this example we are matching some previously defined list named `lst` and seeing if the structure is anything like what we have on lines 3-7. If we take a look at line 7, we will see this pattern `h::t`. Remember our syntax of a list: `e1:t :: e2:t list`. This pattern is just a single value cons to some list of some arbitrary size. Ultimately, as long as a list's size is greater than 1, this pattern would match, but since it's the last item, it will only be reached if the preceding patterns do not match.

6.4.4 Recursive Functions

Knowing all this, we can then make functions that find the head of a list, or the last item of a list, but first remember what I said earlier, there is no looping construct except recursion. So we need to make recursive functions. To make a recursive function is the same as how we construct any other function but we need the `rec` keyword. Let's unalive 2 creatures with 1 weapon:

```
1 (* Assume the list cannot be empty *)
2 let rec tail lst = match lst with
3     [x] -> x
4     |_::t -> tail t
```

First, notice a recursive function is similar to a normal function. We just need the `rec` keyword after the `let` keyword. Next, let's consider the patterns I used. If we assume the list is not empty, then the base case is a list with 1 item. In this case, just return that 1 item. Otherwise, if the list takes the form of `_::t`, or something cons list, then just recursively call `tail` on the rest of the list. Notice that since I did not need the head item, I did not need to bind it to a variable, so I could just use the wildcard character. Another recursive function example: sum up the values in an `int list`.

```
1 let rec sum lst = match lst with
2     [] -> 0
3     |h::t -> h + sum t
```

There are other data types that exist besides lists which you can use and pattern match on. Please refer to the Data Types and Syntax section for more (Section 6.5).

6.5 Data Types and Syntax

6.5.1 Data Types

Basic Types

Data Structures

There are 4 main data structures that exist in OCaml. They are

- Lists
- Tuples
- Variants
- Records

. Each one of these things can be pattern matched and used to construct more complicated data structures. However in my experience I have rarely ever used records.

I talked about lists in an earlier section of this chapter so you can refer there for more info but here are some examples.

```
[1;2;3] (* int list*)
[] (* empty list, Nil, 'a lst *)
[2.0 +. 3.4] (* float list *)
let f x = x + 1 in let g y = y * 1 in [f;g] (* (int -> int) list *)
```

Now that we are refreshed on `lists`, let's talk about tuples. Tuples are ways for us to package data together to be a single 'value' so to speak. This can be useful since functions can only have one return value, so if we need to return multiple pieces of data, a tuple could be the way to go. But enough talking, here is an example:

```
(* tuples.ml *)
(3,4) (* int * int *)
(1,2,"hello") (* int * int * string *)
(* syntax *)
(* (e1:t1,e2:t2,...,ex:tx):t1 * t2 * ... tx *)
```

As you can see tuples are just expressions that comma delimited and placed in parenthesis. Some important things to note is that tuples are of fixed size and their type is dependent on the size and types of the subexpressions. For example, `(3,2)` is an `int * int` tuple which is different than `3,2,1)` which is an `int * int * int` tuple. We can pattern match to break apart tuples by using our match expression.

```
(* tuple-match.ml *)
let t = (1,2)
match t with
|(0,0) -> 0
|(1,1) -> 1
|(1,b) -> b + b
|(a,b) -> a * b
```

The next data type we can talk about are variants. These are similar but not the same as enums. They are ways we can give names and make our own types in OCaml.

```
(* variants.ml *)
type coin = HEADS | TAILS
let x = HEADS (* type is coin, value is HEADS *)
```

These types are then recognized by the rest of OCaml and we can write functions based on these types. To figure out what type you are using, you can use pattern matching.

```
(* variants.ml *)
type coin = HEADS | TAILS
let flip c = match c with HEADS -> TAILS | TAILS -> HEADS
(* flip is a function of type coin -> coin *)
type parity = Even | Odd
let is_even p = match p with Even -> true | Odd -> false
(* is_even is a function of type parity -> bool *)
```

These variants are helpful for just renaming or making data values look pretty. For each of these examples we could have just used bools or ints to represent data. However, variants also allow us to store data in our custom types. Consider the following:

```
(* variants.ml *)
type shape = Rect of int * int | Circle of float
let r = Rect 3 4 (* type is shape, value is Rect(3,4) *)
let c = Circle 4.0 (* type is shape, value is Circle(4.0) *)
```

Here I am saying to make a shape type and that shapes can either be `Rects` which hold `int * int` tuple information, or `Circles` which hold `floats`. We can pattern match to figure out what type we are talking about, and to pull out information.

```
(* variants.ml *)
type shape = Rect of int * int | Circle of float
let r = Rect 3 4 (* type is shape, value is Rect(3,4) *)
let c = Circle 4.0 (* type is shape, value is Circle(4.0) *)
let area s = match s with
Rect(l,w) -> float_of_int l * w (* need to cast this to float so return types match *)
|Circle(r) -> r * r * 3.14
```

This is useful if we want to make Trees or our own lists.

```
(* variants.ml *)
type int_tree = Node of int * int_tree * int_tree | Leaf
let t = Node(4,Node(3,Node(2,Leaf,Leaf),Leaf),Node(5,Leaf,Leaf))
(* tree that looks like
    4
   / \
  3   5
 /
2
*)
```

6.5.2 Syntax

Chapter 7

Higher Order Functions

> Higher Order Functions? More like lower suggestion inabilities
>
> Klef

7.1 Intro

We cover this topic in Ocaml so the examples here will be mostly written in Ocaml. A variation of this chapter written primarily in Python has been made as well.

7.2 Functions as we know them

Let us first define a function. A function is something that takes in input, or an argument, and then returns a value. As programmers, we typically think of functions as a thing that takes in multiple inputs and then returns a value. Technically, this is syntactic sugar for the most part (but that's a different chapter). The important idea now is that we have a process that has some sort of starting values, and then ends up with some other final value.

In the past, functions may have looked liked any of the following:

```
\\ java
int area(int length, int width){
    return length * width;
}
/* C */
int max(int* arr, int arr_length){
    int max = arr[0]
    for(int i =1; i < arr_length; i++)
        if arr[1] > max
            max = arr[i];
    return max;
}

# Ruby
def char-sum(str)
    sum = 0
    str..each_char{|i| sum += i.ord}
    sum
end
```

```
# python
def spam(x):
  return x - 1

(* OCaml *)
let circumference radius = 3.14 *. 2. *. radius

// Rust
fn foo(x:i32){
  let y = x + 2;
  y
}
```

In these functions, our inputs were things like data structures, or 'primitives'. Ultimately, our inputs were some sort of data type supported by the language. Our return value is the same, could be a data structure, could be a 'primitive', but ultimately some data type that is supported by the language.

This should hopefully all be straightforward, a review and pretty familiar. Notice there are 3 (I would say 4) parts of a function. We have the function name, the arguments, and the body (and then I would include the return type or value as well). Again this shouldn't be new, just wanted this here so we are all on the same page.

7.3 Functions as Data

Let's consider the C code:

```
1  int foo = 3;
2
3  int bar(){
4    return 3;
5  }
6
7  int main(){
8    int y = bar() + foo;
9    printf("%d\n",y);
10   return 1;
11 }
```

What exactly is happening here? We could say in line 1, we are allocating a segment in memory, binding that memory address to the human readable name `foo` and then storing 3 in that memory location.

What about lines 3-5? How is `bar` stored in memory? If we consider what is going on in the machine (Maybe recall from 216), then we know that any piece of data is just 1s and 0s stored at some memory address. The variable name helps us know which memory address we are storing things (so we don't have to remember what we stored at address 0x012f or something). When we want to then refer to that data, we use the memory address (variable name) and we retrieve that data. Why should a function be any different?

In this case, we are allocating a segment in memory (in the code part, rather than stack or heap), binding that memory segment to a human readable name `bar`, and then storing the code that represents the function to that location.

We can then treat the `bar` variable like we would treat any other variable. To be clear, they still follow the variable rules that other variables have. `3 + x` is only valid if `x` is an int or similar. So `bar` can only be used where we expect a function (`bar + 3` fails because we are treating `bar` as a number instead of a function).

7.4 Higher Programming

As we said, functions take in arguments that can be any data type supported by the language. A higher order programming language is one where functions themselves are considered a data type. We've seen this in OCaml, but let's take a deeper

look at it now.

Let us consider the following C program:

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

int add1(int x){
  return x + 1;
}
int sub1(int x){
    return x - 1;
}

// return a function pointer
int* getfunc(){
  int (*funcs[2])(int) = {sub1, add1};
  return funcs[rand()%2];
}

// take in a function pointer
void apply(int f(int), int arg1){
  int ret = (*f)(arg1);
  printf("%d\n",ret);
}

int main(){
  int i;
  srand(time(NULL));
  for(i = 0; i < 5; i++){
    apply(getfunc(),3); //playing with pointers
  }
}
```

This program has one function that returns a function pointer and one function that takes in a function pointer. The idea of this is the basis of allowing functions to be treated as data. For most languages, we have the ability to bind variables to data.

```
int x = 3; // C, Java
y = 4 # Ruby
let z = 4.2;; (* OCaml *)
// idea
// variable = data
```

If we consider what is going on in the machine (maybe recall from 216), then we know that any piece of data is just 1s and 0s stored at some memory address. The variable name helps us know the memory address at which we are storing things (so we don't have to remember what we stored at address 0x012f or something). When we want to then refer to that data, we use the memory address (variable name) and we retrieve that data. Why should a function be any different? We previously saw a pointer to a function being passed around, which just means the pointer to a list of procedures that are associated with the function.

So in the case of higher order programming, we are allowing functions to be passed in as arguments or be returned as data.

Thus, we can say that a higher order function is one that takes in or returns another function. We can also avoid all these `void` pointers and casting and stuff in most functional languages like OCaml:

```
(* takes in a function)
let apply f x = f x;;
```

```
(* returns a function *)
let get_func = let add1 x = x + 1 in add1;;
```

7.5 Anonymous Functions

So we just said that we bind data to variables if we want to use them again. Sometimes though, we don't want to use them again, or we have no need to store a function for repeated use. So we have this idea of anonymous functions. It is anonymous because it has no variable name, which also means we cannot refer to it later. The syntax of an anonymous function is

```
(* add 1 *)
fun x -> x + 1
(* add *)
fun x y -> x + y
(* general syntax *)
(* fun var1:t1 var2:t2 ... varx:tx -> e:ty *)
(* has type (t1 -> t2 -> ... -> tx -> ty) *)
```

The difference between `2 + 3` and `let x = 2 + 3` corresponds to the difference between `fun x -> x + 1` and `let x = fun x -> x + 1`. This means that we can do the same thing by doing something like

```
2 + 3                    (* expression by itself, no variable *)
let x = 2 + 3            (* expression then bound to a variable *)
fun x -> x + 1           (* function by itself, no variable *)
let add1 = fun x -> x + 1 (* function bound to variable *)
```

This means `let add1 x = x + 1` is just syntactic sugar of `let add1 = fun x -> x + 1`. This is because OCaml and other functional programming languages are based on this thing called lambda calculus, which is another chapter. But if we think about our mathematical definition of a function, it is something that takes in 1 input and returns 1 output. So if each function should have 1 input, then what about something like `let plus x y = x + y`?

7.6 Partial Applications

Recall a section or something ago when we said that higher order functions can take in functions as arguments and return functions as return values. Consider:

```
let plus x y = x + y
(* int -> int -> int *)
```

We said earlier that functions have types, where the last thing in the type is the return value, and the first few items are the input types. We kinda lied. Let us consider:

```
let plus x y = x + y
(* int -> int -> int *)
let plus x = fun y -> x + y
(* int -> int -> int *)
(* int -> (int -> int) *)
```

This last function does have type `int -> int -> int` but consider what the syntax says. `plus` is a function that takes in an `int` but then returns a function that itself takes in an `int` and returns an `int`. Which means we can actually define plus as

```
let plus = fun x -> fun y -> x + y;;
```

If we can define functions like this then we can do things like

```
let plus = fun x -> fun y -> x + y
let add3 = plus 3 (* returning fun y -> 3 + y *)
add3 5 (* returns 8 *)
```

This is called a partial application of a function, or the process of currying. Not all functional languages support this unless the function is specifically defined as one which returns a function.

It is important to note here that you can only partially apply variables in the order used in the function declaration. That is a function like `let add = fun x -> fun y -> x + y` can only partially apply the x variable: `let add4 = add 4`. This is because we are technically doing something like `let add4 = fun y -> 4 + y`. If we wanted to partially apply the second parameter we would need to do something `let flip f x y = f y x in flip sub`.

To be more clear:

```
let sub x y = x - y
(* same as let sub = fun x -> fun y -> x - y *)
let minus3 = sub 3
(* let minus3 = fun y -> 3 - y *)
(* we cannot partially apply the second argument to sub unless we have a new function *)
(* we could make a sub specific function *)
let minus y = sub y 3
(* but let's make something generic *)
let flip f x y = f y x
(* let flip = fun f -> fun x -> fun y -> f y x *)
let sub3 = flip sub 3
(* let sub3 = fun y -> sub y 3 *)
```

So how does and currying supported language know what the values of variables are? Or how are partially applicated functions implemented? The answer lies with this idea of a closure, something thing that a Ruby Proc is.

7.7 Closures

If you look up the `Proc` object in the Ruby Docs, you will see that they call a Proc a closure. A closure is a way to create/bind something called a context or environment. Consider the following:

```
let and4 w x y z = w && x && y && z
(* and4 = fun w -> fun x -> fun y -> fun z -> w && x && y && z *)
let and3 = add4 true
(* and3 = fun x -> fun y -> fun z -> true && x && y && z *)
let and2 = and3 true
(* and2 = fun y -> fun z -> true && true && y && z *)
```

How does the language or machine know that you want to bind say variable `w` to `true`? To be honest, there is no magic, we just store the function, and then a list of key-value pairs of variables to values. This list of key-value pairs is called an environment. A closure is typically just a tuple of the function and the environment. Visually, a closure might look like the following:

```
let sub x y = x - y
let sub3 = sub 3
(* sub3 may look like
(function: fun y -> x - y, environment: [x:3])
*)
```

Very much like a `Proc` (because a `Proc` is a closure), a closure is not evaluated, or run until it is called. Thus, once made, the closure will not be modified. Thus the following would have no affect:

```
let sub x y = x - y
let x = 3
let sub3 = sub x
let x = 5
sub3 5 (* evaluates to -2 since 3 - 5 = -1 *)
```

Because the environment is not modified, and is evaluated with values that existed at the time of the closure's creation, we say that closures use static scope. This term is used in contrast with dynamic scope, where environment variables get updated to match typically top level variables. That is the above example would return `0` instead of `-2`.

7.8 Common HOFs

Part of the reason why higher order functions (HOFs) are so useful is because it allows us to be modular with out program design, and separate functions from other processes. To see this, consider the following that we say earlier:

```
let sub x y = x - y
let div x y = x / y
let mystery x y = (x*2)+(y*3)
let sub3 y = sub y 3
let div3 y = div y 3
let double y = mystery y 0
```

The functions `sub, div,` and `mystery` are all non-commutative (the order of inputs matter), so if we want to partially apply the second value, we need to write a new function that takes in a value to do so. Alternatively, we can just make a generic function that partially applies the second value so we don't need to ask for any input.

```
let flip f x y = f y x
let sub3 = flip sub 3
let div3 = flip div 3
let double = flip mystery 0
```

Being able to make similarly structured functions into a generic helps makes things modular, which is important to building good programs and designing good software. So the next sections are about common HOFs which will attempt to make a common function structure generic.

7.8.1 Map

Let us consider the following functions:

```
let rec double_items lst = match lst with
[] -> []
|h::t -> (h*2)::(double_items t)

let rec is_even lst = match lst with
[]->[]
|h::t -> (if h mod 2 = 0 then true else false)::(is_even t)

let rec neg lst = match lst with
[] -> []
|h::t -> (-h)::(neg t)
```

All of these functions aim to iterate through a list and modify each item. This is very common need and so instead of creating the above functions to do so, we may want to use this function called Map. Map will *map* the items from the input list (the domain) to a list of new item (co-domain). To take the above function and make it more generic, let us see that is the same across all of them:

```
let rec name lst = match lst with
[] -> []
|h::t -> (modify h)::(recursive_call t)
```

If we think about how we modify h, we will realize that we are just applying a function to h. Since it's the function that changes, we probably need to add it as a parameter. So adding this we should get

```
let rec map f lst = match lst with
[] -> []
|h::t -> (f h)::(map f t)
```

Fun fact: map actually exists in Ruby (`[1,2,3].map!{|x| x+1}`). Either way, in OCaml and other languages without imperative looping structures, this is a common recursive function that is needed and can be used to modify each item of a

list by building a new list of the modified values (recall that everything in OCaml is immutable). Consider the code trace for adding 1 to each item in a `int list`.

```
let add1 x = x + 1 in map add1 [1;2;3]
(*
map add1 [1;2;3] = (add1 1)::(map add1 [2;3])
map add1 []2;3] = (add1 2)::(map add1 [3])
map add1 [3] = (add1 3)::(map add1 [])
map add1 [] = []
map add1 [1;2;3] = (add1 1)::(add1 2)::(add1 3)::[]
map add1 [1;2;3] = 2::3::4::[]
map add1 [1;2;3] = [2;3;4]
*)
```

7.8.2 Fold

While modifying each item in a list is useful, it is not the only common and useful list operation. Consider the following:

```
let rec concat lst = match lst with
[]-> ""
|h::t -> h^(concat t)

let rec sum lst = match lst with
[]-> 0
|h::t -> h+(sum t)

let rec product lst = match lst with
[]-> 1
|h::t -> h*(product t)

let rec ands lst = match lst with
[]-> true
|h::t -> h && (ands t)

let rec length lst = match lst with
[]-> 0
|_::t -> 1+(length t)
```

Here we want to take all the items in a list and return a single aggregate value. Doing this process is called folding and there are two common implementations. To start off, we will define define `fold_right`.

fold_right

So let us find out what each function has in common and then we can figure out what we need to add.

```
let rec name lst = match lst with
[] -> base_case
|h::t -> h operation (recursive_call t)
```

Taking a look at what we need, we need a base case, and we need an operation.

```
let rec fold_r f lst base = match lst with
[] -> base
|h::t -> f h (fold_r f t b)
```

Let us first talk about why it's `f h (fold_r f t b)`. In looking what was the same, we saw that it was `h operator (rec_call t)`. An operator is just a function so we are calling a function with 2 parameters: `h` and the recursive call `fold_r f t b`.

Some of you may also be wondering what about `lenth`? It doesn't use h, it uses a constant 1. My answer to this is to consider the type of the `f`. `f` is a function which takes in 2 parameters, h and the recursive call. I can easily just not use h in the body of `f`. Consider the following code trace:

```
1  let myfunc h rc = 1 + rc in
2  fold_r myfunc [1;2;3] 0
3  (*
4  fold_r myfunc [1;2;3] 0 = myfunc 1 (fold_r myfunc [2;3] 0)
5  fold_r myfunc [2;3] 0 = myfunc 2 (fold_r myfunc [3] 0)
6  fold_r myfunc [3] 0 = myfunc 3 (fold_r myfunc [] 0)
7  fold_r myfunc [] 0 = 0
8  fold_r myfunc [3] 0 = myfunc 3 0 = 1 + 0 = 1
9  fold_r myfunc [2;3] 0 = myfunc 2 1 = 1 + 1 = 2
10 fold_r myfunc [1;2;3] 0 = myfunc 1 3 = 1 + 2 = 3
11 *)
```

Notice here that lines 8-10 are just the stack frames all returning and propagating the return value up as stack frames are being popped off the stack. This also happens in `map` but there's something interesting with `fold` so I wanted to bring attention to it. That is, notice that if we send in a huge list we can potentially get a stackoverflow error or whatever OCaml's equivalent is. We can actually avoid this stack overflow and minimize the number of stack frames needed through the use of the other way to implement fold: `fold_left`. This is the default implementation of fold in most languages afaik so we typically just call this `fold`.

fold_left

See the next section (section 7.9) about why this we can minimize the number of stack frame, but since you already know how fold works, here is `fold_left`

```
1  let rec fold f a l = match l with
2  [] -> a
3  |h::t -> fold f (f a h) t
```

I used the variable a instead of base case or whatever because in this variation, the value is going to act as an accumulator. That is, this value is going to be constantly updated with each recursive call. Again see the next section for more about the accumulator. One thing to note is that this will evaluate the items in the list in the reverse order as `fold_right`. Consider the code trace for `fold_left`, then see them compared together.

```
(* take the sum of the list *)
let add x y = x + y in
fold add 0 [1;2;3;]
(*
fold add 0 [1;2;3] = fold add (add 0 1) [2;3] = fold add 1 [2;3]
fold add 1 [2;3]  = fold add (add 1 2) [3] = fold add 3 [3]
fold add 3 [3] = fold add (add 3 3) [] = fold add 6 []
fold add 6 [] = 6
*)
```

Now to compare the order of `fold_right` and `fold_left` we will use a non-commutative function: subtraction.

```
fold (-) 0 [1;2;3;]
(*
fold (-) 0 [1;2;3] = fold (-) ((-) 0 1) [2;3] = fold (-) -1 [2;3]
fold (-) -1 [2;3]  = fold (-) ((-) -1 2) [3] = fold (-) -3 [3]
fold (-) -3 [3] = fold add ((-) -3 3) [] = fold add -6 []
fold (-) -6 [] = -6
*)
(* compare this to fold_right *)
fold_r (-) [1;2;3] 0
```

```
(*
fold_r (-) [1;2;3] 0 = (-) 1 (fold_r (-) [2;3] 0)
fold_r (-) [2;3] 0 = (-) 2 (fold_r (-) [3] 0)
fold_r (-) [3] 0 = (-) 3 (fold_r (-) [] 0)
fold_r (-) [] 0 = 0
fold_r (-) [3] 0 = (-) 3 0 = 3
fold_r (-) [2;3] 0 = (-) 2 3 = -1
fold_r (-) [1;2;3] 0 = (-) 1 -1 = 2
*)
(* -6 != 2 *)
```

How interesting.

7.9 Tail Call Optimization

I was going to make this it's own chapter, but then had logistical questions so for now I decided against it and so I will just put this in the HOF chpater for some reason.

Let us take a trip back to our 216 days when we learned about stack frames and function calls. One thing I have noticed is that students get weird around recursion but I want you to consider the following

```
1  int fact1(int x){
2    if (x == 1)
3      return 1;
4    return -1
5  }
6  int fact2(int x){
7    if (x == 2)
8      return 2 * fact1(x-1);
9    return -1
10 }
11 int fact3(int x){
12   if (x == 3)
13     return 3 * fact2(x-1);
14   return -1
15 }
16 int fact4(int x){
17   if (x == 4)
18     return 4 * fact3(x-1);
19   return -1
20 }
```

Suppose we are on line 18. To evaluate what is returned, we have to call `fact3`, wait for it's return value, and then use that return value by multiplying it by 4. This is no different than it's recursive equivalent

```
1  int fact4(int x){}
2      if (x == 1)
3          return 1;
4      if (x <= 4)
5          return x * fact4(x-1);
6      return -1;
7  }
```

The only difference is instead of calling a different function, waiting for it's return value, then using it's return value, we are instead calling ourself, waiting for a return value, then using that return value.

Great, so now that we know how recursion works, recall how a stack frame is created and pushed onto the memory stack when a function is called and then popped off the memory stack then the function returns. So the difference between something like the non-recursive `fact4` and the recursive `fact4`, is which function is being pushed to the stack.

So Consider what the stack looks like for the recursive `fact4` if we call `fact4(3)`

```
//Bottom of Stack//
3 // push argument on stack
---start of fact4(3) stack frame---
return 3 * fact4(2)
---end of fact4(3) stack frame---
2 // push argument on stack
---start of fact4(2) stack frame---
return 2 * fact4(1)
---end of fact4(2) stack frame---
2 // push argument on stack
---start of fact4(1) stack frame---
return 1
---end of fact4(1) stack frame---
```

Here we are pushing on stack frames when we call the recursive call. Then when finally get to out base case, we can then start popping values off. So popping off the textttfact4(1) call would make the stack look like

```
//Bottom of Stack//
3 // push argument on stack
---start of fact4(3) stack frame---
return 3 * fact4(2)
---end of fact4(3) stack frame---
2 // push argument on stack
---start of fact4(2) stack frame---
return 2 * 1
---end of fact4(2) stack frame---
```

When you learned recursion, you probably learned about return values being propagated when teh function returns and this is how you can communicate values from one stack frame to another. This is definitely what happens, but notice that with something like recursive Fibonacci, you will get stack frames being added exponentially and you will get something like a `stackoverflow` error.

```
int fib(int x){
    if(x <= 1)
        return 1;
    return fib(x-1) + fib(x-2);
}
```

Here the number of stack frames increase at a rate of 2^x since each call to `fib` will push 2 more `fib` stack frames.

I think we can all agree that `Stackoverflow` errors are not good and if we can avoid them, we should. One way to avoid this is to use **tail call optimization** which would be something a compiler would use to optimize your code. To talk about tail call optimization, let us first talk about what the actual issue is.

The issue is that there are too many stack frames on the stack and then we run out of memory. There is 2 ways we can solve this issue: 1) add more memory or 2) pop things off the stack. The first solution doesn't really fix the issue, since memory is finite and we can just ask for something like `fib(10000000)`. The second solution has an issue because we need the old stack frames to exist. However, let us consider why we need the old stack frames.

In the previous example, we needed the old stack frame because before we could return, we needed the return value of a different stack frame.

```
//Bottom of Stack//
// calling fact4(3)
---------------------------------
return 3 * fact4(2)
//cannot return here since we need to first calculate fact4(2)
---------------------------------
return 2 * fact4(1)
```

```
//cannot return here since we need to first calculate fact4(1)
--------------------------------
return 1
--------------------------------
```

We said earlier that one way to pass in data from one stack frame to another is via the return value. However this is just communication from the callee to the caller. We can pass information from the caller to the callee by via argument values. So let consider this new factorial function:

```
int fact(int n, int a){
    if(n<=1)
        return a;
    return fact(n-1, n*a);
}
```

Notice that I added a new argument, a. This new parameter will allow the caller to send in data to the callee during the recursive call. Consider the following trace:

```
//Bottom of Stack//
// calling fact(3,1)
--------------------------------
// fact(3,1)
return fact(3-1,3*1) // fact(2,3)
--------------------------------
// fact(2,3)
return fact(2-1,2*3) // fact(1,6)
--------------------------------
// fact(1,6)
return 6
--------------------------------
```

Notice here that we get the same value, passing in the work of each stack frame into the next recursive call. What this means is that we no longer need to wait for the recursive call to finish, we can instead pop off stack frames once the recursive call happens.

```
//Bottom of Stack//
// calling fact(3,1)
--------------------------------
// fact(3,1)
return fact(3-1,3*1) // fact(2,3)

// we don't need the fact(3,1) stack frame so pop it off and push on fact(2,3) in it's place

//Bottom of Stack//
// calling fact(2,3)
--------------------------------
// fact(2,3)
return fact(2-1,2*3) // fact(1,6)

// we don't need the fact(2,3) stack frame so pop it off and push on fact(1,6) in it's place

//Bottom of Stack//
// calling fact(1,6)
--------------------------------
// fact(1,6)
return 6

// got the correct return value
```

So why is this called a tail call optimization and how to we make sure we are tail recursive? To answer this question let us look at the syntax of these recursive calls.

```
int nontailfact(int x)
    if (x == 1)
        return 1;
    return x * nontailfact(x-1);
}

int tailfact(int n, int a){
    if(n<=1)
        return a;
    return tailfact(n-1, n*a);
}
```

Where the one major difference is the number of arguments, tail optimization does not care about this. Remember that we care about the behavior of the recursive call. So if we notice the syntax around the recursive call, we can say that we care about what the last thing being calculated is during the recursive call. In the `nontailfact` the last thing being calculated is `x * nontailfact(x-1)`. In the `tailfact`, the last thing being calculated is `tailfact(n-1,n*a)`. This is purely a syntactical (visual) thing so we say that any statement that could be the last thing executed is in `tail position`. If the recursive call is in tail position, then we can take advantage of tail-call optimization.

Let us consider the tail position of some OCaml statements.

```
3
4
"a"
(* all of these statements are in tail position, since they are the last thing being
    evaluated *)

2 + 3
4 * 5
(* here 2,3,4,5 are not in tail position. The last thing calculated is 2*3, so we say the
    entire expression is in tail position. This is a tad confusing so let's see something
    clearer *).

[2+3;5*4;0-1]
(* here the last thing being evaluated is the creation of the list. So despite 2*3 being the
     last expression being evaluated, we still need to create the list so the entire
    expression is again in tail position *)

let x= 3 * 4 in x + 4
(* the last thing here is x+4 so the expression x + 4 is in tail position *)

let x = 3 + 4 in let x = 6 in 7
(* consider the syntax we used for a let binding: let v = e1:t1 in e2:t.
Here x = v, e1 = 3 + 4, and (let x = 6 in 7) is e2. Here at the top level, (or in broadest
    context), the expression in tail position is e2 or (let x = 6 in 7). If we changed our
    context to be more "zoomed in" or "jump in instead of jump over" then things in tail
    position would be just 7 *)
```

Again this is purely a syntactical thing which depends on the context of which parts of the expression will we consider. In an earlier section, we talked about `fold_right` and `fold_left`. They both do the same thing(ish), but one of them is tail recursive, and the other is not.

Chapter 8

Finite State Machines

Finite State Machines? More like infinite town devices

Cringe

8.1 Introduction

So far we have talked about the language features that languages may have. However, now we want to start talking about how we can take features from one language and implement them in another. A naive approach may be something like making a library or some wrapper functions. For a simple example, maybe I wanted to add `booleans` to C. I can just write a `#define` macro for `1` and `0` which we name as `true` and `false`. For a more complicated example if I wanted to add pattern matching in C, then maybe I create a `struct` called `data` which can hold any value which can be pattern matched and a function: `void* match(data* value, int (**patterns)( data* ), void* (**exprs)(data*))` which takes in a piece of data to match, a series of functions that return true if the value matches it, and a series of functions that return some value[1]. This way is terrible and so the typical way to add something is by changing the compiler (Or if we want to go one step further, let's design our own language, which means we need to make a new compiler-HAH!).

8.1.1 Compilers

While this is not CMSC430: Compilers, we need to setup the basis of compilation. We will talk about this more in depth in a future chapter, but here's a quick overview. A compiler is a language translator (typically some higher level programming language to assembly). To translate one language to another, we need to do the following:

- break down the language to bits that hold information
- take those bits and figure out how to store that information in a meaningful way
- take the stored information and map it to the target language
- generate the target language.

The best way to break down the language is to use regular expressions. However, what if your language doesn't have regular expressions? Simple: let's implement regular expressions in a way that we don't need to compile.

8.1.2 Background - Automata Theory

Imagine that we want to create a machine that can solve problems for us. Our machine should take in a starting value or values, a series of steps, and then give us output. Depending on when and who you took CMSC250 with, you already did this.

[1] See Appendix A for a rough implementation //TODO

A circuit or logic gate is the most basic form of this. If our input is values of true and false (1 and 0), let us put those inputs into a machine that *and*s, *or*s and *negate*s to get an output value.

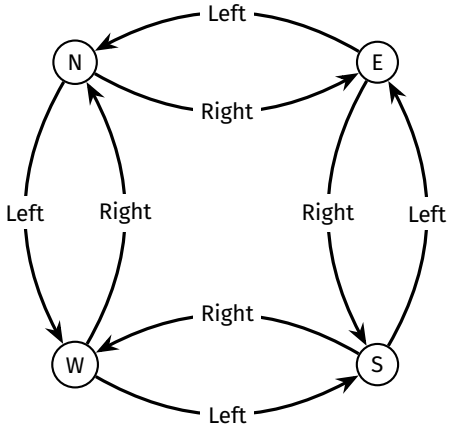
The issue here is we do not have any memory. We cannot refer to things we previously computed, but only refer to things we are inputting in each gate. Once we add a finite amount of memory, we can accomplish a whole lot more and we get what we call finite automata (FA). I use finite automata interchangeably with finite state machine (FSM). Typically, FA is used in the context of abstract theory and FSM is used with the context of an actual machine, but they both refer to the same thing.

Once we start adding something like a stack (infinite memory), we get a new type of machine called push-down automata (PDA) where we theoretically have infinite memory, but we can only access the top of the stack. Lifting this top-only read restriction, we get what is known as a Turing machine[2]. As formalized in the Church-Turing thesis, any solvable problem can be converted in a Turing Machine. A Turing machine that creates or simulates other Turing machines is called an Universal Turing Machine (UTM). Fun fact: Our machines we call computers are UTMs).

All of this is to say that a compiler wants to output a language that is Turing complete, one which can be represented by a Turing machine. Regular expressions on the other hand, describe what we call regular languages, and regular languages can be represented by finite automata. So we will start with FSMs, but know that when we get to compilation, we will need something more.

8.1.3 Finite State Machines

Let's start by modeling a universe and breaking it down to a series of discrete states and actions. Let us suppose that my universe is very small. There is just me, a room and a compass. Suppose I am standing facing north in this room. Let's call this state N. When facing north, I have two options: turn right 90° and face East, or turn left 90° and face West. Let's give these states some names: states E and W respectively. From each of these new positions (facing west or facing east), I could turn left or right again and either end up facing back north or facing south. Let's give the state of facing south a name: S. If I create a graph that represents all possible states and actions of the universe, I could create a graph that looks like:



This graph represents a finite state machine. A physical machine can be made to do these things, but for the most part, we will emulate this machine digitally. We typically define a FSM as a 5-tuple:

- A set of possible actions
- A set of possible states
- a starting state
- a set of accepting states
- a set of transitions

The set of transitions is the set of edges, typically defined as 3-tuple (starting state, action, ending state). To be clear: this is a graph. A transition is an edge, and a state is a node. We haven't seen what a starting or accepting state is, but we will see those in the next section.

[2] Initially called an 'a-machine' or atomic machine by Alan Turing.

The important takeaway from the example above is **Based on where I am (which state), and what action occurs (which edge I choose), I can tell you where I will end up**. So, given an input and a series of instructions, I can give you an output (sound familiar?). For example, if I start at state N, and my instructions are to go left, left, right, left, left, right, right, I can traverse my path ($N- > W- > S- > W- > S- > E- > S- > W$) to know where I am and return it (My output is W here).

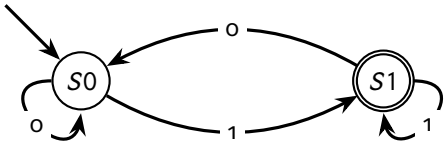
8.2 Regex

So we did this whole thing with graphs and talked about what a machine is and a single-example use case. Let's talk about another use case: regular expressions. For regular expressions, we define a FSM as a 5-tuple very similarly as what we previously had, but instead of actions, we have letters of the alphabet.

So, we have:

- the alphabet (Σ), which is a set of all symbols in the regex
- a set of all possible states (S)
- a starting state (s_0)
- a set of final (or accepting) states (F)
- a set of transitions (δ)

To be clear on types, $s_0 \in S$ and $F \subseteq S$. This is because a FSM can only have 1 starting state (no more, no less), but any number of accepting states (including 0). In the previous example, we can understand the starting state to be N, but we don't really have any accepting states. Let us see an example of a FSM for the regular expression `/(0|1)*1/`.



This machine represents the regular expression `/(0|1)*1/`. Recall that a regular expression describes a set of strings. This set of strings is called a language. Examples of strings in the language described by the regular expression `/(0|1)*1/` would be `"1"`, `"10101"`, and `"0001"`. When we say that a FSM accepts a string, it means that after entering at the starting state (denoted by an arrow with no origin) and traversing the graph after looking at each symbol in the string, we end up in an accepting state (states denoted by a double circle). Let's see an example.

Given the above FSM, suppose we want to check if the string `"10010"` is accepted by the regex. We start out in state $S0$ since it has the arrow pointing to it as the starting state. We then look at the first character of the string: `"1"` and consume it. If we are in state $S0$ and see a `"1"`, we will move to state $S1$. We then look at the next second of the string (since we consumed the first one): `"0"` and consume it. Since we are in state $S1$, if we see a `"0"`, then we move to state $S0$. We then proceed to traverse the graph in this manner until we have consumed the entire string. The traversal should look something like

```
S0 -1-> S1 -0-> S0 -0-> S0 -1-> S1 -0-> S0
```

Since we end up at state $S0$, and $S0$ is not an accepting state (it does not have a double circle), then we say this machine (this regular expression) does not accept the string `"10010"`. Which is true, this regex would reject this string.

On the other hand if traversed the graph with `"00101"`, our traversal would look like

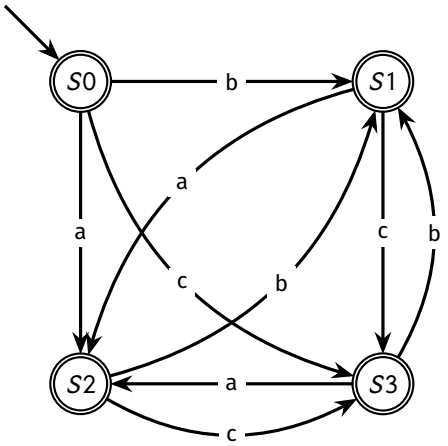
```
S0 -0-> S0 -0-> S0 -1-> S1 -0-> S0 -1-> S1
```

and we would end up in state $S1$ which is an accepting state. So we could say that the machine (the regular expresion) does accept the string `"00101"`.

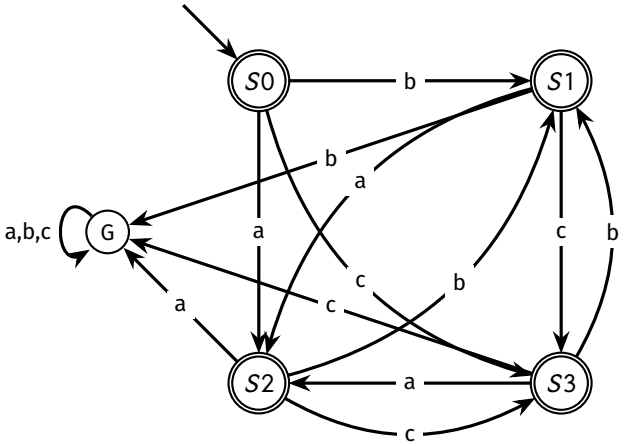
One final thing: a FSM for regex only will tell you if a string is accepted or not[3]. It will not do capture groups, will not tell you if the input is invalid, it will only tell you if the string is accepted (in the case of invalid input, it will tell you the string is not accepted).

8.3 Deterministic Finite Automata

All FSMs can be described as either deterministic or non-deterministic. So far we have seen only deterministic finite automata (DFA). If something is deterministic (typically called a deterministic system), then that means there is no randomness or uncertainty about what is happening (the state of the system is always known) [4]. For example, given the following DFA:



Now this graph is missing a few states (one really). What happens when I am in state $S1$ and I see a `"b"`? There is an implicit state which we call a `"garbage"` or `"trash" state`. A trash state is a non-accepting state where once you enter, you do not leave. There is an implicit one if you are trying to find a transition symbol or action which does not have output here. That is, there is an edge from $S1$ to the `garbage state` on the symbol `"b"`. There are also transitions to the garbage state from $S2$ on `"a"` and $S3$ on `"c"`. If we really wanted to draw the garbage state in, we could like so (but there really is no need to do so):



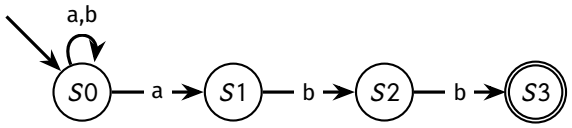
Regardless if we indicate a trash state or not, no matter which state I am in, I know exactly which state I will be in at any given time.

[3] that is, if the string is in the language the regular expression describes. Any language a regular expression can describe is called a Regular Language

[4] Determinism in philosophy is about if there is such a thing on free will and I would definitely recommend reading David Hume's and David Lewis's take on causality

8.4 Nondeterministic Finite Automata

The other type of FSM is a nondeterministic finite automata (NFA). Nondeterministic in math terms means that is something that is some randomness or uncertainty in the system. A NFA is still a FSM, the only difference is what are allowed as edges in the graph. There are 2 of them. Let's talk about one of them now. Consider the following FSM:

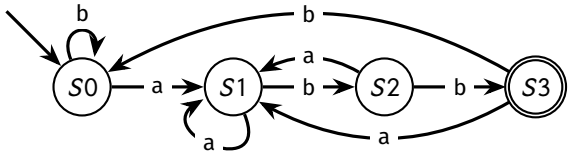


This machine represents the regular expression `/(a|b)*abb/`. There is still a starting state, transitions, ending states, all the things we see for a FSM. However, there is something interesting when we look at the transitions out of $S0$. If I am looking at the string `"abb"`, then when I am traversing, do I go from $S0$ to $S1$ or do I loop back around and stay in $S0$? In fact, there are two ways I could legally traverse this graph:

```
S0 -a-> S1 -b-> S2 -b-> S3
S0 -a-> S0 -b-> S0 -b-> S0
```

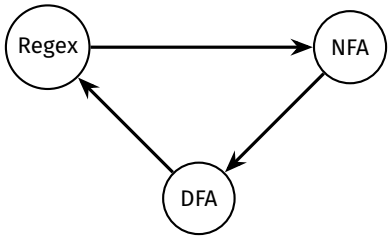
Since the traversal of the graph is uncertain, we call this non-deterministic. To check acceptance for an NFA, we have to try every single valid path and if at least one of them ends in an accepting state, then we accept the string. Since we have to check all possible paths, you can imagine that this is quite a costly operation on an NFA. Additionally, all NFAs have a DFA equivalent. So why use an NFA?

Let us first consider why we are using a FSM. We want to implement regex. To convert from a regular expression to a DFA can be difficult. Consider the above NFA for `/(a|b)*abb/`. Now consider the following DFA for the same regex:



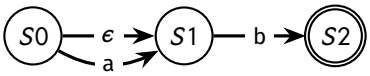
It is much easier to go from a regular expression to an NFA than it is to go from a regular expression to a DFA. Additionally, NFAs, because they cen be more condensed, are typically more spatially efficient than their DFA counterpart. However, there is of course a downside: NFA to regex is difficult, and checking acceptance is very costly. However, NFA to DFA is a one time cost and its less costly to check acceptance on a DFA. Additionally, going from a DFA to a regular expression is much easier. Now keep in mind, technically all DFAs are NFA, but not all NFAs are DFAs.

To visualize this, we typically draw the following triangle



We will talk about how to convert between all of these in a bit, but before we get too far ahead of ourselves, we need to consider the other difference an NFA has over a DFA: epsilon transitions.

An ϵ-transition is a "empty" transition from one state to another. If we think of our graph as one where the edges transitions are the cost to traverse that edge, then an ϵ-transition is an edge that does not cost anything to traverse (it does not consume anything). Consider the following NFA:



If I wished to check acceptance of the string `"b"`, then my traversal may look like:

```
S0 -ε-> S1 -b-> S2
```

Whereas my traversal of the string `"ab"` may look like:

```
S0 -a-> S1 -b-> S2
```

Knowing this, you can see that this machine represents the regular expression: $/a?b/$.

8.5 Regex to NFA

Now that we know what a FSM, NFA and a DFA is, then we can loop back around to our initial goal: implementing regular expressions. In order to do this, we will of course need to build an NFA. To do so, we need to think about the structure of a regular expressions. That is, we need to consider what the grammar of a regular expression. We will talk about grammars in a future chapter, but a grammar is basically rules that dictate what makes a valid expressions. Here is the grammar for a regular expression:

$$\begin{aligned} R \rightarrow \quad & \emptyset \\ & |\epsilon \\ & |\sigma \\ & |R_1 R_2 \\ & |R_1 | R_2 \\ & |R_1^* \end{aligned}$$

All this says is that any Regular expression is either

- something that accepts nothing ($\emptyset$)
- something that accepts an empty string (ϵ)
- something that accepts a single a single character (σ)
- a concatenation of 2 Regular expressions ($R_1 R_2$)
- One regular expression or another regular expression ($R_1 | R_2$)
- A Kleene Closure of a regular expressions (R_1^*)

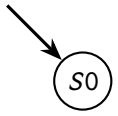
To convert from a regular expression to a NFA, all we need to figure out how to represent each of these things as an NFA. Since this grammar is recursive, we will start with the base cases, and then move on to the recursive definitions.

8.5.1 Base Cases

There are three base cases here: $\emptyset, \epsilon, \sigma$. Let's look at each of these.

The $\emptyset$

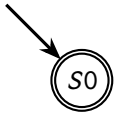
The empty set is a regex that accepts nothing (the set of strings (the language) it accepts is empty). This machine can be constructed as just the following:



Even if Σ has characters in it, we just have a single state, and upon seeing any value, we get a garbage state.

The empty string (ϵ)

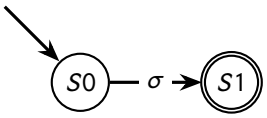
The regular expression that only accepts the empty string means the set of accepted strings is $\{""\}$. This set is not empty so it is different from $\emptyset$. This machine can be constructed as the following:



Even if Σ has characters in it, we just have a single state, and upon seeing any value, we get a garbage state since we are not accepting any strings of with a size greater than 0.

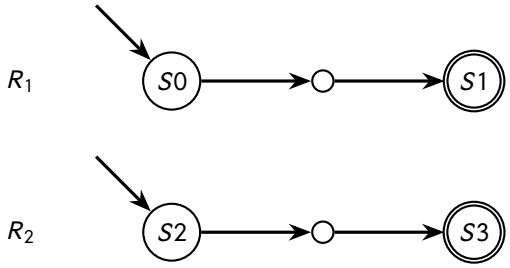
A single character σ

The last base case is a regular expression which accepts only a single character of the alphabet. So if $\Sigma = \{"a", "b", "c"\}$, then we are looking for a regular expression that describes only `/a/`, `/b/`, or `/c/`. We call a single character σ. This machine can be represented in the following manner:



8.5.2 Concatenation ($R_1 R_2$)

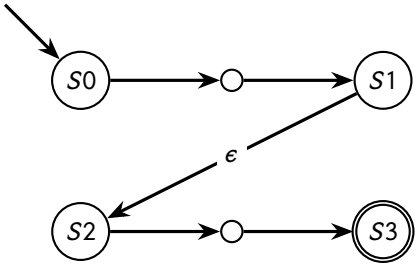
Now that we have our base cases, we can begin to start showing how to do a recursive operation. Aside from the $\emptyset$, each base case has a starting state and an accepting state (sometimes these are the same state). Now since the $\emptyset$ is empty, all the recursive definitions cannot rely on it, so we don't need to really include it as a base case for these recursive calls. Hence, let us assume we have some previous regular expressions R_1 and R_2 that have a starting state and an accepting state.



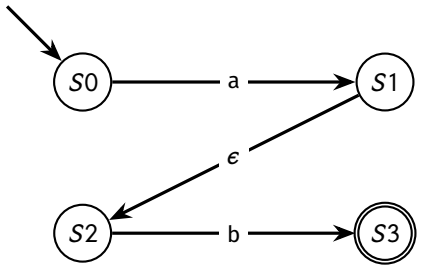
Now we don't know what regular expression R_1 and R_2 are, just that they have 1 starting state, and 1 accepting state. The small, unlabeled nodes here just represent any internal nodes could exist (if any).

To concatenate these two together, it means that if L_1 is the language corresponding to R_1 and L_2 is the language corresponding to R_2, then we are trying to describe L_3 which can be represented as $\{xy | x \in L_1 \wedge y \in L_2\}$. For example, if R_1 is `/a/` and R_2 is `/b/`, then $L_1 = \{"a"\}$ and $L_2 = \{"b"\}$. This means that $L_3 = \{"ab"\}$.

So to take our previous machines, and create a new machine which represents our concatenation operations, we can do so by looking at what our new final states are, and how we get an ordering. Our new machine should have 1 final state which should be the same as our R_2 machine, and should have a way to get from R_1 to R_2 without costing us anything. By implementing these two steps, we get the following machine:



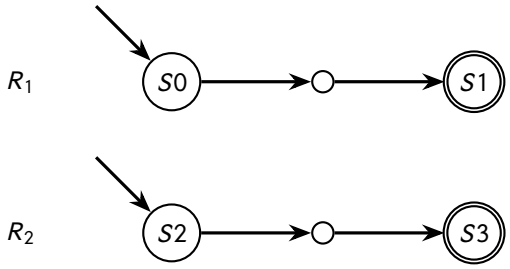
To take our previous example of R_1 being /a/ and R_2 being /b/, when we want to concatenate these machines we get the following machine:



Now, I would say that we are done at this point, as we have a machine that accepts only the concatenated string, with one starting state and one final state (having only one final state is not a restriction of a FSM, but having only one allows us to inductively build our machines here). If we really wanted to, we could optimize the machine a little bit, but it is not necessary.

8.5.3 Branching ($R_1|R_2$)

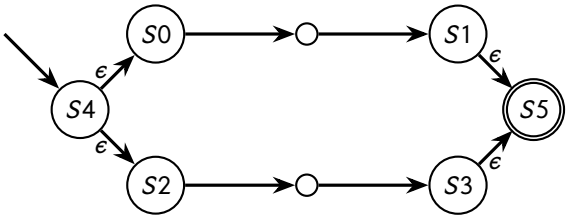
Branching or union is the next recursive definition and requires a bit more work than our concatenation. Again, let us assume that we have some previous regular expressions R_1 and R_2 that have a starting state and an accepting state.



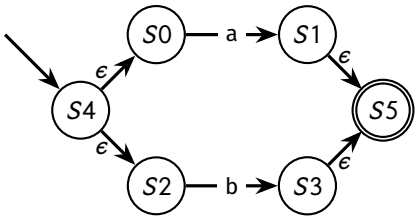
Again we don't know what regular expressions R_1 and R_2 are, just that they have one starting state and one accepting state.

To union two regular expressions together it means that if L_1 is the language corresponding to R_1 and if L_2 is the language corresponding to R_12, then we are trying to describe L_3 which can be represented as $\{x|x \in L_1 \vee x \in L_2\}$. For example if R_1 is /a/ and R_2 is /b/, then $L_1 = \{"a"\}$ and $L_2 = \{"b"\}$. This means that $L_3 = \{"a", "b"\}$.

So to take our previous machines and create a new machine which represents our union operation, we can do so by considering what it means to traverse the graph such that either previous machine is valid. Again, to keep our inductive properties, we need one starting state and one accepting state. Here is where the tricky part comes. We need to make sure that both R_1 and R_2 are accepted with a single accepting state, as well as making sure we can traverse R_1 or R_2 with only 1 start state. The easiest way to do so is by making use of ϵ-transitions with 2 new states. Here is the resulting machine:



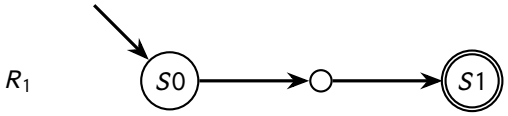
This new machine still has one starting state, and one accepting state which means we can inductively build larger machines, and the ϵ-transitions allow us to chose either path or go to the accepting state without consuming anything. To take our previous example of R_1 being /a/ and R_2 being /b/, when we want to union these machines we get the following machine:



Again, we could choose to optimize this machine, but it's not necessary.

8.5.4 Kleene Closure (R_1^*)

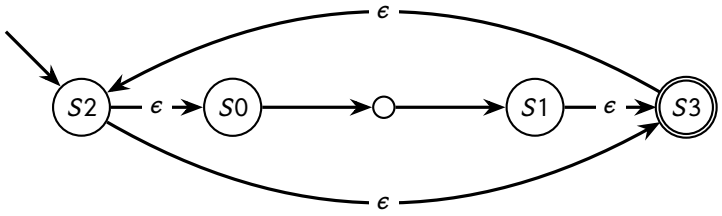
Kleene closure gives us the ability to repeat patterns infinitely many times and is the last recursive definition of a regular expression. Despite having the ability to infinitely repeat, it will look very similar to our union machine. Additionally, this is the only recursive definition that does not rely on two previous regular expressions, so here we only need to assume that we have some previous regular expressions R_1 that has a starting state and an accepting state.



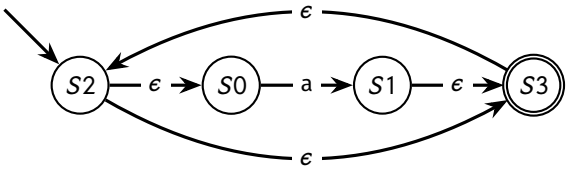
Again we don't know what regular expression R_1 is, just that it has one starting state and one accepting state.

The Kleene Closure of a language, is just analogous to the language's regular expression with the $*$ operator. That is, if L_1 is the language corresponding to R_1 then we are trying to describe L_2 which can be represented as $\{x | x = \epsilon \vee x \in L_1 \vee x \in L_1L_1 \vee x \in L_1L_1L_1 \vee \dots\}$. For example, if R_1 is `/a/` then $L_1 = \{\text{"}a\text{"}\}$ and we are looking for `/a*/` or $L_2 = \{\text{""}, \text{"}a\text{"}, \text{"}aa\text{"}, \text{"}aaa\text{"}, \dots\}$.

So if we take our previous machine, we need to consider how we can accept the empty string as well as any number of repeats of a regular expression. The trick for this is in the definition. We are essentially 'or'ing together the same regular expression repeatedly. So will need to designate a new start state and a new ending state. Doing so will result in the following machine:



Here is where the ϵ-transitions become really important. To accept the empty string, we just use an ϵ-transition to move from $S2$ to $S3$. For repeated values, we can just use the ϵ-transitions from $S3$ to $S2$. Let's look at the previous example of R_1 being `/a/` and seeing the resulting Kleene closure, but also how we would traverse it. The machine would look like:



If I wanted to accept the empty string my traversal would look like

```
S2 -ϵ-> S3
```

If I wanted to accept `"a"`, then my traversal would look like

```
S2 -ϵ-> S0 -a-> S1 -ϵ-> S3
```

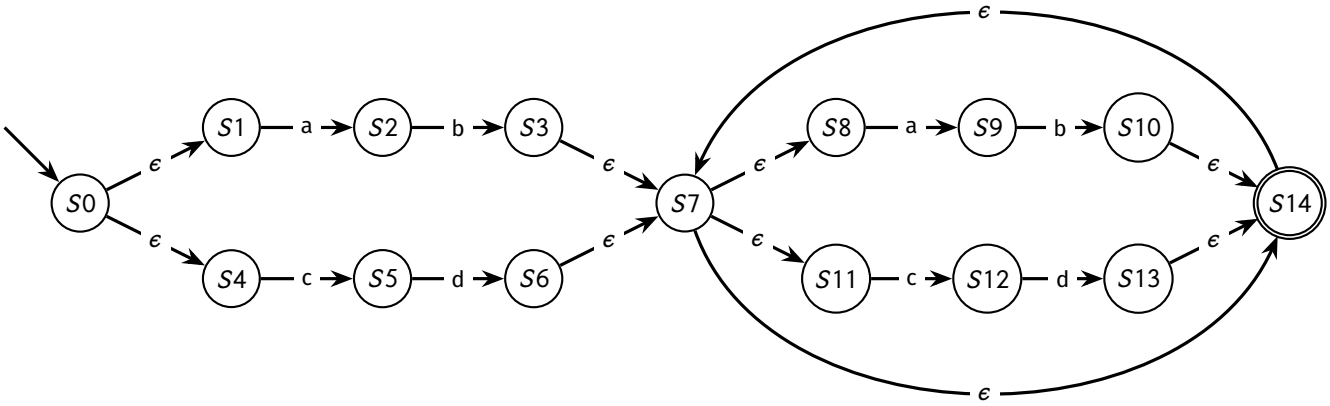
If I wanted to accept `"aa"`, then my traversal would look like

```
S2 -ϵ-> S0 -a-> S1 -ϵ-> S3 -ϵ-> S2 -ϵ-> S0 -a-> S1 -ϵ-> S3
```

We can of course optimize this machine, but again it is not necessary.

8.5.5 Example

For a quick example, if we wanted to make the NFA for the regular expression: `/(ab|cd)+/` the machine (with a few optimizations due to space) would look like:

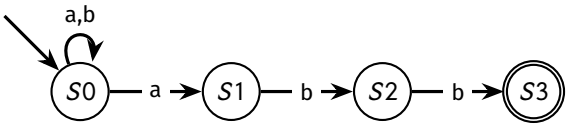


We could optimize this even further, but not really needed. Additionally to see without any optimizations and for a step-by-step, you can see Appendix B (//TODO).

8.6 NFA to DFA

Now that we know how to convert from a regular expression to a NFA, we should talk about to how to convert from a NFA to a DFA. The reason being is that checking for acceptance on an NFA can be really costly and typically you will be calling `accept` multiple times on a machine. So instead of calling `nfa-accept` *n* times, which is a costly operation, you should convert the NFA to a DFA (which is still costly, but done once), so you can then call `dfa-accept` *n* times which is a very cheap operation.

So lets start out by considering the difficulty of `nfa-accept`. Consider the following NFA:

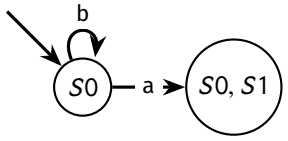


When we want to check acceptance, we need find all possible paths and check if at least one accepts it. That is when checking if the machine accepts `"aabb"`, we need to check all of the following paths:

```
S0 -a-> S1 -a-> Garbage
S0 -a-> S0 -a-> S0 -b-> S0 -b-> S0
S0 -a-> S0 -a-> S1 -b-> S2 -b-> S3
```

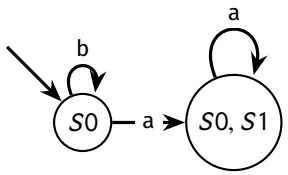
Then since one of them ends in the accepting state, then we can say this machine accepts this string. Notice that we are essentially doing a depth-first-search here which can be terrible if we got an NFA that has a Kleene closure because we get an infinite depth. Additionally, the crux of the problem is that we have no idea which state I am in when I first see the `"a"`. I could either go from `S0 -a-> S0` or I could go from `S0 -a-> S1`. This uncertainty is what makes an NFA non-deterministic. The solution here is to create a new state which represents this uncertainty.

To demonstrate this idea, let's add a state that says "I don't know if I am in state $S0$ or $S1$". Notice this will only happen when we start by looking for an `"a"`. Additionally, when we start with a `"b"`, we know that we have to stay in state $S0$ (that is, if we are in state $S0$ and see a `"b"`, we can only go to $S0$. There is no uncertainty here. But if we are in state $S0$ and see a `"a"`, we could be in $S0$ or $S1$).



Now while we have a new state that shows possible states I could be in after seeing a `"a"`, I then need to figure out what to do next. That is, if I am in this new state $S0, S1$, then what happens if I see a `"a"` or a `"b"`?

If this state shows where I could possibly be, then we need to consider both possibilities [5]. So going back to the original NFA, if I am in state $S0$ and see a `"a"`, I could go to state $S0$ or $S1$. Additionally, (looking at the origianl NFA), if I am in state $S1$ and I see a `"a"`, then I can't go anywhere but the garbage state. So not including the garbage state, we can say that regardless of being in $S0$ or $S1$, if I see an `"a"`, then I have to either be in state $S0$ or $S1$. Well we already have a state that represents this possibility so we can just add the following transition:



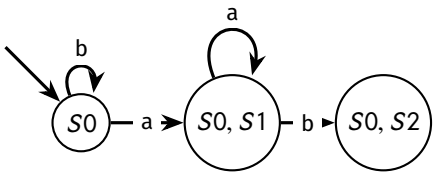
If this is confusing, consider the following logical argument:

$$\begin{array}{ll} & p \Rightarrow q \\ & s \Rightarrow r \\ & p \vee s \\ \hline \therefore & q \vee r \end{array}$$

From CMSC250 we know this is a valid logical argument (known as constructive dilemma). The same applies here. If $S0$ and `"a"` leads to $S0$ and $S1$, and $S1$ and `"a"` leads nowhere (except the garbage state) then we will and up in either $S0$ or $S1$ (or the garbage state).

$$\begin{array}{ll} & S0 \Rightarrow \{S0, S1\} \\ & S1 \Rightarrow \varnothing \\ & S0 \vee S1 \\ \hline \therefore & \{S0, S1\} \cup \varnothing = \{S0, S1\} \end{array}$$

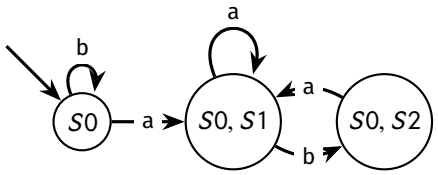
Anyway, that was a slight sidetrack. We next need to consider if we are in $S0, S1$ and we see a `"b"`. The above logic applies. If I am in state $S0$ and see a b (looking at the original NFA), then I will end up in state $S0$. If I am in state $S1$ of the original NFA and see a b, then I will end up in state $S2$. It is uncertain which state I will be in though so let's add a new state that represents being in $S0$ or $S2$.



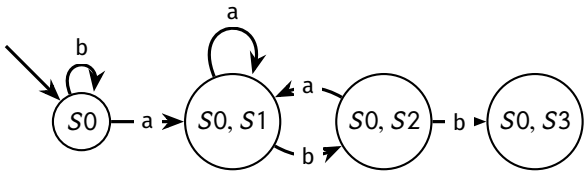
[5] In "laymen's" terms, we are in quantum superposition, that is we are in both $S0$ and $S1$ at the same time

But now the issue continues, if I am in state $S0$, $S2$, and see a symbol, I do not know where I should go. SO let's continue with considering if I was in $S0$ or $S2$ and seeing either a `"a"` or a `"b"`.

If I am in state $S0$ and see a `"a"`, then I am either in $S0$ or $S1$. If I am in state $S2$ and see a `"a"`, then I can go nowhere (except the garbage state). So if I am in either state $S0$ or $S2$ and see a `"a"`, then I will end up in either $S0$ or $S1$ (or garbage). Let us add this transition.

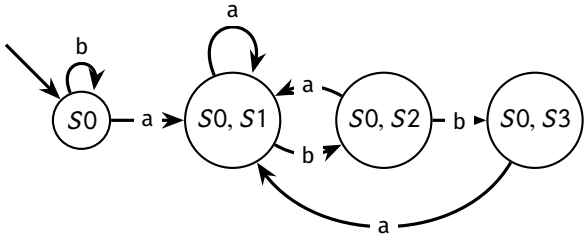


Now if I am in state $S0$ and see a `"b"` then I can only go to state $S0$. If I am in state $S2$ and see a `"b"`, then I can only go to state $S3$. So if I am in either state $S0$ or $S2$, then I will end up in either $S0$ or $S3$. Here is another place of uncertainly so let us add this new state with transition.

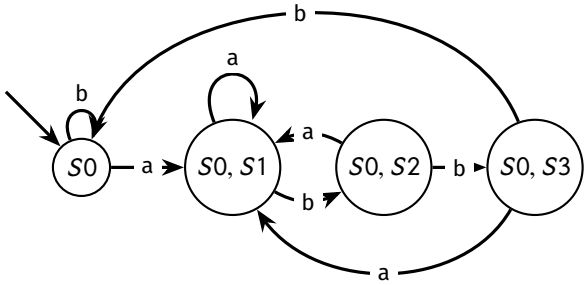


Again, the same issue arises. If I am in state $S0$, $S3$, what happens if I see a `"a"` or `"b"`? Well we will have to calculate this like we did with the other states.

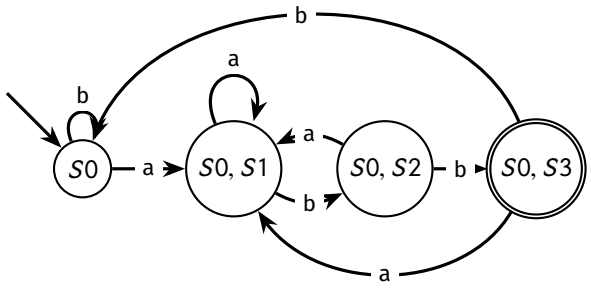
If I am in state $S0$ and see a `"a"`, then I could either be in $S0$ or $S1$. If I am in state $S3$ and see a `"a"`, then I can go nowhere (except a trash state). So if I am in either state $S0$ or $S1$, then I can only end up in $S0$ or $S1$. Let us add this transition.



Now if I am in state $S0$ and see a `"b"` then I can only go to state $S0$. If I am instate $S3$ and see a `"b"`, then I cannot go anywhere (except garbage). So if I am in state $S0$ or $S3$, if I see a `"b"`, I can only really go to $S0$. So let us add this transition:



Now notice that this time around, we did not add any new states and we know where we want to go from each state. That is, there is no ϵ-transitions, and no state has multiple outgoing edges on the same symbol. By not having these two things, we have created a DFA from our initial NFA. There is just one final step: which states should be our accepting states? If the whole thing is based on possibility being in a state, then it should follow that any state which represents a possible accepting state should in turn, be an accepting state. In our original NFA, $S3$ was the only accepting state, so we look at all states of this DFA and mark any state which represents the possibility of being in $S3$ as an accepting state.



If you also go back a few pages, this machine is identical to the DFA we said corresponded to our NFA. Wild.

8.6.1 NFA To DFA Algorithm

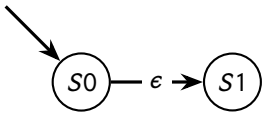
So let us convert what we just did to an algorithm.

To do so, we will need to define 2 subroutines: ϵ-`closure` and `move` as well as define our NFA. Let us use the same FSM defnition we have been using: $(\Sigma, S, s_0, F, \delta)$. Let us give some types as well:

- Σ: `'a list`, a list of symbols
- S: `'b list`, a list of states
- s_0: `'b`, a singe state ($s_0 \in S$)
- F: `'b list`, a list of state we should accept ($F \subseteq S$)
- δ: `('b * 'a * 'b) list`, a list of transitions from one state to another, (`(source, symbol, destination)`).

ϵ-Closure

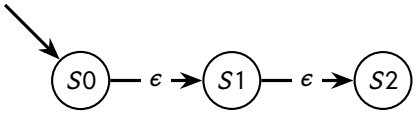
Now to our functions. Recall that we want to figure out which states can be grouped together. ϵ-closure is a function that helps figure this out in terms of ϵ-transitions. THe previous example did not have any ϵ-transitions but consider:



If I am in state $S0$, I could also possibly be in state $S1$ as well. So ϵ-closure if a function that helps us figure out where can we go using only ϵ-transitions. Now the term closure should give us a hint as to what we want to do. We want to figure out what states are closed upon ϵ-transitions. It is important to note that any state can reach itself via an ϵ-transition. So here is the type of ϵ-closure:

- `e-closure: (NFA -> 'b list -> 'b list)`, given a list of states, return a list of states reachable using only ϵ-transitions

To see an example, let us consider the following machine:



If I were to call ϵ`-closure nfa [S0]` I should get back `[S0;S1;S2]`. The best way to do so is by iterating through δ and checking where you can go to anywhere in the input list. Then recursively calling ϵ-closure on the resulting list. That is:

```
e-closure nfa [So]
// Looking at So I can only go to So and S1 via an epsilon transition
[So;S1]
// looking at So I can go to So and S1, looking at S1 I can go to S1 and S2
[So;S1;S2]
// looking at So I can go to So and S1, looking at S1 I can do to S1 and S2,
// looking at S2 I can go to S2
[So;S1;S2]
// I got no new states, my output matches my input, I am done
```

Here since, my output matches my output then I can return this list and be done. This type of algorithm is called a fixed point algorithm.

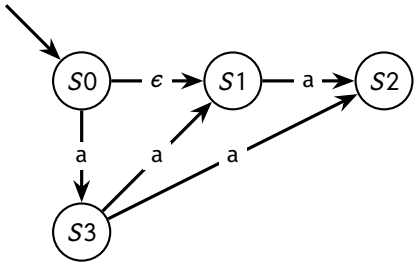
The actual pseudocode is something like:

```
NFA = (alphabet, states,start,finals,transitions)
e-closure(s)
  x = s
  do
    s= x
    x = union(s,{dest|(src in s) and (src,e,dest) in transitions})
  while s!= x
  return x
```

On the other hand, `move` is going to just see where you can do based on a starting state and a symbol. It's type is

- `move: (NFA -> 'a -> 'b -> 'b list)`, given a state and a symbol, return a list of states I could end up in.

It is important to note that you should not perform ϵ-closure at any point during a move. For an example, consider the following machine:



If we were to call `move "a" S0`, then we should get back `[S3]`. Yet if we were to call `move "a" S3` we should get back `[S1;S2]`. This one is pretty straightforward. Just iterate through δ to figure out what your resulting list should be.

NFA to DFA Pseudocode

Now that we have everything defined, need to take the process we had and create an algorithm. Going back to the NFA to DFA example, on each step we had to figure out where we could be upon each symbol in the alphabet. So our pseudocode should look like:

```
NFA = (a, states, start,finals,transitions)
DFA = (a, states, start,finals,transitions)
visited = []
let DFA.start = e-closure(start), add to DFA.states
while visited != DFA.states
  add an unvisited state, s, to visited
  for each char in a
    E = move(s)
    e = e-closure(E)
    if e not in DFA.states
```

```
      add e to DFA.states
    add (s,char,e) to DFA.transitions
DFA.final = {r| r∈ DFA.states and ∃ s ∈ r and s ∈ NFA.final}
```

For a full in-depth example, see Appendix C //TODO

8.7 DFA to Regex

Chapter 9

Languages

I'm a programmar that is not pro-grammar

Cliph

9.1 Introduction

We have now done a ton in regex, but the issue with regular expressions is that they can only do so much. Regular expressions are great when talking about regular languages, but programming languages (and spoken languages) are not regular languages, so we need something more expressive. Let us consider what exactly we need to describe a language.

9.2 Context-Free Grammars

Recall that a language is just a set of strings. We used regular expressions to tell you how to construct the strings in the set. An alternative way is to give a recursive definition of the set. Recall from 250 what a recursive set is (in this case the set of positive multiples of 3):

$$S = \begin{cases} 3 \\ x \in S \implies x + 3 \in S \end{cases}$$

We can do the same thing to describe sets of strings (although we use a slightly different notation). This is called a grammar. We will be going over context-free grammars (CFGs) as opposed to context-sensitive grammars (CSGs). For example, let us take the CFG we saw for regular expressions:

Any regular expression can be expressed as a string that is in the following set

$$\begin{aligned} S \to \quad & \epsilon \\ & |\sigma \\ & |SS \\ & |S|S \\ & |S^* \\ & |(S) \end{aligned}$$

Any valid regular expression sentence follows the above pattern.

Let's break this down and understand exactly what it means. Here, S is called a Non-terminal because S could be a variety of things. On the other hand symbols like $\epsilon, \sigma, ^*, |, (,)$ are what we call terminals. Grammars also have what is called productions: rules about what non-terminals can be. This example is honestly not the best for these terms, but we will see an example soon, and we will revisit these terms.

The important part right now is that this grammar describes all strings that represent a regular expression. Very much like we can use finite automata to represent a regular expression and show that the regular expression accepts a string, we

can derive a string from a grammar using substitution to show that a string is grammatically correct (and hence belongs in the language the grammar describes).

For example, the regular expression `/ab*/` can be described using the following derivation:

$$\begin{aligned} S &\rightarrow SS \\ &\rightarrow aS \\ &\rightarrow aS^* \\ &\rightarrow ab^* \end{aligned}$$

Those who have taken a linguistics or hearing and speech sciences class, or even an English class, may know that “grammar" typically refers to the order in which words need to be for the sentence to make sense. That is still true here.

Unlike `/ab*/`, the regular expression `/*b/` cannot be derived from the above grammar, so we claim it to be grammatically incorrect and not part of the language.

Now, English is complicated and has a lot of rules, but consider the following simplified grammar for English.

$$\begin{aligned} S &\rightarrow NP\ VP \\ NP &\rightarrow \text{pronoun} \\ &\quad |\text{proper\_noun} \\ &\quad |\text{det } AN \\ AN &\rightarrow \text{adj } AN \\ &\quad |\text{noun} \\ VP &\rightarrow \text{verb} \\ &\quad |\text{verb } NP \end{aligned}$$

Let us revisit our terms from earlier to break this down. Non-terminals are symbols that represent other symbols. Conventionally, we give them uppercase letters. Sometimes, the letters mean something; sometimes they are just alphabetical. In this case, they mean something (NP stands for noun phrase, VP for verb phrase, and AN for adjective noun).

- **Terminals**: `pronoun, proper_noun, det, adj, noun, verb` are all terminals. Unlike the previous example where a lowercase letter was a symbol, these all stand for larger sets of things (This can be confusing, so in this course we will typically only be using symbols like in the first example).
- **Non-terminals** S, NP, VP, AN are all non-terminals. We know this because they are all uppercase, and each has a production rule associated with it.
- **Production**: a production rule tells us all the things a non-terminal can be. For example, $S \rightarrow NP\ VP$ is a production rule. It states that any sentence S consists of a noun phrase NP followed by a verb phrase VP.
  $AN->$ adj $AN|noun$ says that any AN phrase is an adjective followed by another AN phrase, or it is just a noun.

Using this grammar, we can still derive if a sentence is grammatically valid in English. For example, if I had a sentence like "The child ran the race", then I could say that this sentence is grammatically correct and should be in the set of valid English sentences. Before I show the derivation, let us make sure we know our parts of speech:

- "The" is a determiner (det) because it determines the reference of something. Some other examples are "every", "a", "some", and "each".
- "child" and "race" are nouns since they fall under the category of a person, place, or thing (or idea).
- "ran" is a verb since it describes an action.

Knowing all this, let us now show the derivation:

$$\begin{aligned} S &\rightarrow NP\ VP \\ &\rightarrow \text{The (det) } AN\ VP \\ &\rightarrow \text{The child (noun) } VP \\ &\rightarrow \text{The child ran (verb) } NP \\ &\rightarrow \text{The child ran the } AN \\ &\rightarrow \text{The child ran the race} \end{aligned}$$

The sentence S is a noun phrase (NP) followed by a verb phrase (VP). In the example above, the first noun phrase is going to use the third definition of a noun phrase: det AN. "The" is the determiner. The following AN then uses the second definition of being just a noun, which in this case is "child". So the noun phrase is "The child". The verb phrase is going to use the second definition of a verb phrase: verb NP. The verb here is "ran", and the noun phrase is going to be "det AN". In this case, the determiner is again "the" and the AN is just a noun: "race". More on this in a bit.

9.3 Designing Grammars

We said that a grammar describes a set of strings, but it is more expressive than regular expressions. This means that any regular expression can be expressed as a CFG but also that CFGs can get around some restrictions that regular expressions have. Let us start with operations that are supported by regular expressions.

9.3.1 Regular Expressions Supported

Let us start with our 3 base cases.

- $\emptyset$: If the language is empty, meaning it is a set containing no strings, then the CFG should reflect that as well. The CFG can still be represented as $\emptyset$, which is the null (empty) set

- ϵ: If the regular expression accepts the empty string, then the CFG can just have a single production.

$$S \rightarrow \epsilon$$

- σ: If the regular expression is just a single character, we can have our CFG reflect that character in a single production.

$$S \rightarrow \sigma$$

  I will say though that in larger grammars, we typically describe sentences or statements with words, so if we have a list of words, we can do the same thing. For example, we can have something like

$$S \rightarrow \text{Cliff}$$

Once we have the base cases (shown above), we can talk about the recursive definitions: concatenation, branching, and kleene closure:

- **Concatenation**: If we wish to concatenate two things together, we can just push them together with either non-terminals or just the string you expect. For example, the corresponding CFG for the regular expressions /ab/ would be

$$S \rightarrow \text{ab}$$

  Alternatively you could do something like

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

  This can be helpful if you have branching (next bit), or just a sentence where you want to force one thing to come before (eg. adjective before noun).

- Branching: If we want to branch, we can use the same symbol we used in regex |. For example, the grammar for a greeting could be

$$S \rightarrow \text{Hello|hi}$$

  You can put each option on a new line if you have the space, but either way is valid.

- Kleene Closure: For allowing repeated values, we can just utilize the recursive property these sets have. For example, the corresponding CFG for the regex /a∗/ is

$$S \rightarrow aS|\epsilon$$

. CFGs also let us have a shortcut for something like /a+/

$$S \rightarrow aS|a$$

We can also use this to repeat whole words:

$$\begin{array}{ll} S \rightarrow & \text{This is a } T \text{ sentence} \\ T \rightarrow & \text{very } T|\text{long} \end{array}$$

9.3.2 Not supported by Regular Expressions

We said that CFGs are more expressive which means we can say more with a CFG than we can with regular expressions. So let us think of some of the restrictions we had with regular expressions. We could not look forward more than one character at a time, and we could never reference what we previously saw. So would couldn't do things like balanced parenthesis. This is all in thanks to the recursive nature of CFGs.

Consider the previous CFG:

$$\begin{array}{ll} S \rightarrow & \text{This is a } T \text{ sentence} \\ T \rightarrow & \text{very } T|\text{long} \end{array}$$

We have a sentence where we have something known ("This is a ") followed by non-terminal (T) which is then followed by some other known string ("sentence"). By allowing for this ability to look at both before and after the non-terminal, we can do things like balance parenthesis, or have relative distinct values.

For something like having a balanced values on either side (parenthesis or palindromes), we can just put the values on either side of the non-termainal.

Balanced parenthesis surrounding "a"	$S \rightarrow (S)\|\text{a}$
Palindromes of "a", "b", and "c"	$S \rightarrow \text{a}S\text{a}\|\text{b}S\text{b}\|\text{c}S\text{c}\|\epsilon$

For having relative number of values we are a tad restricted to a few characters that are relative to each other, but supposed we want a string with the some number of "a"s followed by the same number of "b"s. Our notation for this is $a^n b^n$. The following grammar would allow for that:

$$S \rightarrow aSb|\epsilon$$

We can also do distinct relative numbering like $a^n b^{2n}$:

$$S \rightarrow aSbb|\epsilon$$

Or even $a^n b^m$, $m \geq n$

$$\begin{array}{ll} S \rightarrow & \text{a}S\text{b}|T \\ T \rightarrow & \text{b}T|\epsilon \end{array}$$

Sometimes the order doesn't even matter. If I wanted a string that had the same number of "a"s and "b"s in any order then I could do something like:

$$S \rightarrow S\text{a}S\text{b}|S\text{b}S\text{a}|\epsilon$$

I can also do a string that has an unequal amount of "a"s and "b"s in any order:

$$\begin{array}{ll} S \rightarrow & A|B \\ A \rightarrow & C\text{a}A|C\text{a}C \\ B \rightarrow & C\text{b}B|C\text{b}C \\ C \rightarrow & \text{a}C\text{b}C|\text{b}C\text{a}C \end{array}$$

9.3.3 A basic Grammar

Putting all this together, I could create a rudimentary language that describes basic algebraic expressions.

$$\begin{aligned} A &\rightarrow \quad A + A | A - A | A * A | A / A | N \\ N &\rightarrow \quad number \end{aligned}$$

This grammar is okay because it allows for strings like `"2 + 1 + 0 - 4 * 9 / 3"`. However, this grammar does not allow for things like `"(4-30)*-5`, which of course is allowing order of operations to be expressed. We can just easily modify this expression by adding our parenthesis rule we talked about:

$$\begin{aligned} A &\rightarrow \quad A + A | A - A | A * A | A / A | (A) | N \\ N &\rightarrow \quad number \end{aligned}$$

Now from a compiler/interpreter stand point these this grammar still has some issues, but we will talk about all of this between the next section and the Parsing chapter.

9.4 Modeling Grammars

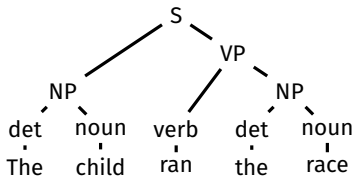
Now that we know what a CFG is and what they represent, we need to discuss how we can model them into something useful (since typically you need to do something with a language and not just know what they are and how they work).

Now one of the leading theories in linguistics and psychology (that I know of) is that we store grammar and the like as a tree in our heads. Regardless if I am up to date or not in the linguistics field, this is what we will be using to model our grammars and sentences in compsci.

Recall that a grammar tells you the structure of a language, so the tree should tell us this as well. We do this by our recursive definition. Consider our basic English Grammar

$$\begin{aligned} S &\rightarrow \quad NP\ VP \\ NP &\rightarrow \quad \text{pronoun} \\ &\quad\ \ | \text{proper\_noun} \\ &\quad\ \ | \text{det noun} \\ VP &\rightarrow \quad \text{verb} \\ &\quad\ \ | \text{verb } NP \end{aligned}$$

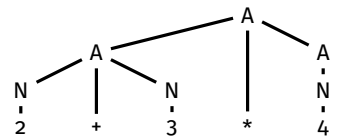
If we take the same sentence we always used: "The child ran the race", we can represent this sentence as a tree thanks to our grammar:



If we consider our basic algebraic expression grammar we can also model sentences with it:

$$\begin{aligned} A &\rightarrow \quad A + A | A - A | A * A | A / A | (A) | N \\ N &\rightarrow \quad number \end{aligned}$$

For example "2 + 3 * 4" can be modeled as:



Technically this tree represents the following derivation:

$$\begin{aligned} A \rightarrow & \quad A * A \\ \rightarrow & \quad A + A * A \\ \rightarrow & \quad N + A * A \\ \rightarrow & \quad 2 + A * A \\ \rightarrow & \quad 2 + N * A \\ \rightarrow & \quad 2 + 3 * A \\ \rightarrow & \quad 2 + 3 * N \\ \rightarrow & \quad 2 + 3 * 4 \end{aligned}$$

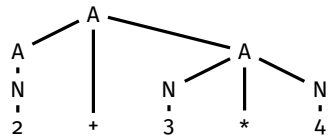
This derivation I got via something we call a "left hand derivation". That is, we will substitute the left most variable for a recursive definition. Consider the right hand derivation for the same tree:

$$\begin{aligned} A \rightarrow & \quad A * A \\ \rightarrow & \quad A * N \\ \rightarrow & \quad A * 4 \\ \rightarrow & \quad A + A * 4 \\ \rightarrow & \quad A + N * 4 \\ \rightarrow & \quad A + 3 * 4 \\ \rightarrow & \quad N + 3 * 4 \\ \rightarrow & \quad 2 + 3 * 4 \end{aligned}$$

Notice that using a left hand or right hand derivation does not change the tree (but if we did more, it would impact the way the tree is build. However, notice we could have used the following left hand derivation instead:

$$\begin{aligned} A \rightarrow & \quad A + A \\ \rightarrow & \quad N + A \\ \rightarrow & \quad 2 + A \\ \rightarrow & \quad 2 + A * A \\ \rightarrow & \quad 2 + N * A \\ \rightarrow & \quad 2 + 3 * A \\ \rightarrow & \quad 2 + 3 * N \\ \rightarrow & \quad 2 + 3 * 4 \end{aligned}$$

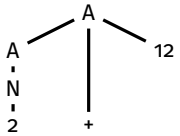
This tree would look like:



Both of these trees and derivations are both valid when substituting the leftmost variable for a definition of A, so we call this grammar ambiguous. A grammar is ambiguous when there are two valid left hand derivations. A grammar can also be ambiguous when where are 2 valid right hand derivation.

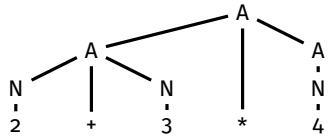
Now that we have some tree model, we need to discuss what we do with these trees. These trees have a proper name as well: parse trees. We will get into parsing in a future chapter, but ultimately should we want to try and obtain meaning from a the tree we need some sort of tree traversal algorithm.

In this case, a post order traversal would probably be helpful here. Consider the second variation to the tree we had. If I wanted to calculate the right-most A, then I would need to figure out what two values my sub children were before I said "3-4". Then I could recursively figure out subtrees until I get to the root. That is I would go from the above tree to the below tree:

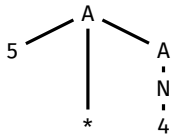


Which would then have the two subtrees be evaluated to 2 and 12 respectively, which we would then multiply to get 24 as the final result.

However, consider the first variation of the tree. If we simplified in this manner we would get the following:



Simplified down to:



And finally to:

20

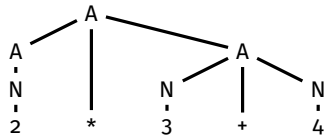
Notice that we get two separate values due to the ambiguity.

There are two ways to help solve ambiguous grammars: we could try and figure out more the grammar and restrict how we go about traversing through the tree, or we can just change the grammar a bit.

For example, much of the ambiguity is because we don't know which path the variable should be when given something. We can fix this by adding separate non-terminals:

$$\begin{aligned} A &\rightarrow \quad N + A | N - A | N * A | N / A | N | (A) \\ N &\rightarrow \quad number \end{aligned}$$

Now we know that any expression must start with a number and not an expression. So constructing a tree of "2∗3+4 could only result in the following:

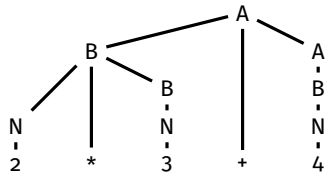


The final issue here is that when we do our traversal, we are still doing addition before multiplication. In order to fix this, we can either mandate that all operations be put in parenthesis or we can change the grammar to have precedence.

The idea of precedence is that we do things of higher precedence closest to the base case as possible. That is notice that when multiplication is closest to the bottom of the tree, we are getting the correct computation. So let us modify our grammar to have precedence. We use the same sort of trick we did for concatenation, we add more non-terminals since we need to figure out non-terminals before terminals. We should get:

$$\begin{aligned} A &\rightarrow \quad B + A | B - A | B \\ B &\rightarrow \quad N * B | N / B | N \\ N &\rightarrow \quad number | (A) \end{aligned}$$

Given this grammar we can only get the following tree:



Chapter 10

Interpreters and Compilers

> Parsing helps us figure out what all these squiggles on the page mean
>
> Cliff

10.1 Introduction

We know how a grammar describes a language. Let us consider the following language which I will call Math-ew:

$$\begin{aligned} E &\rightarrow \quad +NE|-NE|*NE|/NE|N \\ N &\rightarrow \quad 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

Math-ew describes simple mathematical expressions. Some sentences in this language include `"+ 3 4"` and `"- 3 * 4 + 5 2"`. We will use Math-ew throughout this chapter so feel free to refer back to it.

Now that we have a language to work with, the next question is how do we go from something like `"+ 3 4"` to what we can assume to be the correct value, 7? That is, in `utop` or `irb`, why is it that when we enter something like `"3 + 4"` we get back 7?

This is all the work of an interpreter. I personally call this a compiler, but the connotation is important.

10.2 Compilers/Interpreters

Consider what happens when we take a file like `"funs.ml"` and then run `ocamlc funs.ml`. It is ultimately the same thing that happens if we ran something like `gcc prog.c` or `javac Program.java`. We take a text file [1] and compile it down to machine code and then we get some program we can run. Practically though we take the text file and convert it to an assembly file (technically another text file) which an assembler then converts to the appropriate machine code.

All of this is to say we think of compilers as something that takes our code and creates a program, but this is practically incorrect. A compiler is a language translator. So you could theoretically create a Java to C compiler, or a Ruby to Java compiler [2].

An interpreter on the other hand takes a text file and returns a value. `irb` is an interpreter since it does not make assembly or machine code, but instead gives back a value. I would argue this is also a compiler because I could just define the target language as something like

$$S \rightarrow value$$

. However maybe what I should say is that both interpreters and compilers are translators, but they translate in 2 different ways. A compiler translates by making machine code and converting an entire program to an analogous program in a different language. An interpreter translates by converting one statement at a time and evaluating these statements to a value.

[1] The only difference between `"funs.ml"` and `"funs.txt"` is the file name (which is arbitrary).

[2] See `https://pandoc.org/`

Regardless of if we are talking about a interpreter or a compiler, typically there are three things needed to make a translator:

- lexer: converts text file to a list of tokens
- parser: takes list of tokens and creates an intermediate representation (Typically a tree)
- evaluator/generator: takes the intermediate representations and either evaluates to a value, or generates analogous code in a different language.

We will talk about all of these things with an example for Math-ew written in C.

10.3 Lexing

When you are reading this, you are looking at squiggly lines made up of ink or pixels and being literate, you are able to make meaning of these squiggly lines. Like what a superpower: you can look at squiggly lines and then gain knowledge from said squiggly lines.

Lexing is analogous to figuring out what the words are. Consider the following sentence:

Your tongue does not fit comfortably in your mouth.

You see some squiggly lines and then get upset that you are now thinking how you should position your tongue. Magic. But lexing is just the process of figuring out what words are in the sentence (and maybe what type they are). That is, you split up the sentence into nine words: `"Your"`, `"tongue"`, `"does"`, `"not"`, `"fit"`, `"comfortably"`, `"in"`, `"your"`, and `"mouth"`. You may even tag certain words as a noun or verb or determiner. However, notice that we said it's just figuring out what the words are. Given the string

"green the truck"

You still recognize there are three words, `"green"`, `"the"`, and `"truck"`. Despite this being grammatically incorrect, you still were able to figure out these words. That is exactly what a lexer does.

When creating a new language, you would typically have a list of words which should be allowed in the language. In Math-ew we have operator words ("+", "-", "*", and "/") and digit words ("0", "1", "2", ... "8", "9"). So we should have something like

```
type token = Plus|Sub|Mult|Div|Num of int
(*
If we wanted, we could do something like
type token = Operation of string|Num of int
*)
```

We then want a function that takes a string and returns a list of tokens. We could do this by doing something like the following:

```
let rec lex str =
    if str = "" then [] else
    if String.sub str 0 1 = "+" then
        Plus::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "-" then
        Sub::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "*" then
        Mult::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "/" then
        Div::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "0" then
        Num(0)::(lex (String.sub str 1 ((String.length str) - 1)))
    ...
    else if String.sub str 0 1 = "9" then
        Num(9)::(lex (String.sub str 1 ((String.length str) - 1)))
    else lex (String.sub str 1 ((String.length str) - 1))
```

This is not the most efficient way to do this. This also assumes you will be fed valid input (a non-malicious user). I would recommend looking at the `Str` library which lets you use regular expressions in Ocaml.

However, this does work as a lexer.

```
let rec lex str =
lex "1 + 2" = [Num(1); Plus; Num(2)]
lex "1 2 +" = [Num(1); Num(2); Plus]
lex "+ 1 2" = [Plus; Num(1); Num(2)]
```

Again, notice that our lexer does not check if the string matches the grammar. All it does it check if all the words in the string are valid. In this particular lexer, we skip over any unwanted words but may not always be the desired behavior.

10.4 Parsing

Now that we have a list of tokens and know that we have a list of valid words (tokens), we need to make sure our sentence is grammatically correct. That is, if we have the string

green the truck

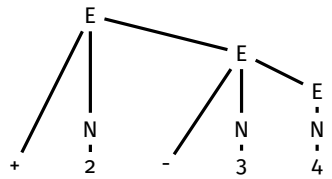
You lex this correctly and recognize all three words (`"green"`, `"the"`, `"truck"`) are valid words in the English language, but you notice this is grammatically incorrect. This is what the parser does. Check if things are grammatically correct or not, while also storing the sentence in an intermediate form, whether it be a tree, code, or something else.

This is typically done to make the evaluator or generating step easier, but from a linguistic point, we gain information just by having something be grammatically correct. Consider the sentence `"I like purple"` and `"I like purple shirts"`. By being grammatically correct, we know that in the first sentence `"purple"` is a a noun, whereas in the second sentence `"purple"` is an adjective. It would be harder to identify this if things were not grammatically correct.

Additionally there are many types of parsers that exist so depending on the grammar or what you want to prioritize using a different parser may be useful . I will be using a Left-to-right Left-derivation parser, that looks ahead by 1 token at a time or a LL(1) parser. A LL(1) parser is a type of a Recursive Decent Parser (RDP). This means that we will be using a left most derivation when checking the grammar, reading from left to right. The lookahead by 1 just means we will be looking one token at a time (to decide what production to use if needed).

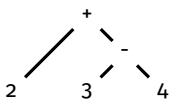
Hence why we may need to change our grammar to match what our parser is capable of. Regardless of which parser you use, you probably want to have a different data structure of representing the sentence. When talking about CFGs, we used a tree which is what we will use here. We will be using a Abstract Syntax Tree (AST) which will just make a tree based off the content of the language. We could make a parse tree instead which makes a tree based off the symbols of our language, but parse trees focus more on non-terminals (internal nodes) vs terminals (leaves) whereas ASTs focus more on content of the string.

Let us consider the following Math-ew sentence `"+ 2 - 3 4"`. The parse tree would look like:



Notice this is more closely aligned with the grammar. Where E is either N or some operator followed by two other E values.

For an AST, we care more about what is occurring. Here is an AST for the same sentence:



Either option is a valid output for a parser, but for more complex grammars, we sometimes don't care about all tokens (like white space or {) or we can represent how some of the tokens are used via the structure of the tree. For example If

we had parenthesis to show order of operations, the above AST would not need to include them since the structure of the tree shows that we want to subtract 4 from 3 before we add 2. Hence a parse tree could be replaced with something more useful.

So if I wanted to create an AST, then I have to consider how to define and design my tree. The nodes in my tree will either be a number (leaf) or a operation with the subtrees being children of the node. Translating this to code looks like:

```
type ast = Int of int
          |Add of ast * ast
          |Minus of ast * ast
          |Times of ast * ast
          |Divide of ast * ast
```

Now we need to make the parser. For Math-ew (and typically other recursive languages), we need to consider the fact that for a sentence like `"- 2 * 2 3"`, that the character `"2"` is a stand alone sentence. However we cannot recursively call here despite the fact that $E \longrightarrow N$. That is we run into a small issue:

```
let rec parser tok_list = match tok_list with
[] -> failwith "Math-ew does not support empty strings"
|Plus::t -> let next_expr = parse t in Add(next_expr, ?)
...
```

We have no idea what should be in the spot of the question mark. Since `parse t` would parse the rest of the list and still have something left over (namely `[Mult; Num(2);Num(3)]`).

We can solve this problem in 2 ways, either we find some way to figuring out our remaining tokens or we can consider how our language is structured. Let's do the latter since it's harder.

In our language, we know that any operator will be followed by a number or an expression. Hence, we can mimic our grammar with our parser like so:

```
let rec parse tok_list = match tok_list with
Num(x) -> Int(x)
|Plus::Num(x)::t -> Add(Int(x), parse_E t) (* + N E *)
...
```

Here, we can break down each branch as an operator, followed by a number, followed by the remainder. Technically though we are looking ahead by 2 tokens so this is a LL(2) parser. We can easily fix this in the following way:

```
let rec parse tok_list = match tok_list with
Num(x) -> Int(x)
|Plus::t -> let h::t = t in Add(Int(h),parse t)
...
```

A slight difference, but syntactically shows that we are only looking 1 token ahead at a time instead of 2. We can expand on this more in appendix D //TODO.

Notice that this also will only match on gramatically correct sentences. So if our sentence was something like `"+ - 2 3 4"` we would not match and we would probably throw an error. Which means that now all we have left is to take meaning from the sentence.

10.5 Evaluating/Generating

Now that we have a parser that generates a tree of some sort, now we need a way to traverse through the tree to compute a final value (at least for an interpreter). We need to create meaning from our AST. So suppose that we have a sentence like:

Colorless green ideas sleep furiously

This contains all valid words in the English lexicon (our lexer says ok), and it is grammatically correct (our parser would make a tree) but in English this means nothing. The evaluater's job is to make sure this can be done. We helped this process by designing our AST in a way that we can easily traverse where each node has meaning.

That is if we consider the above AST, notice that we just need to perform a post order traversal to compute a value. That is, at the root, we need to process our left subtree and then our right subtree, and then we can add our values together. This can very easily be modeled with the following code:

```
let rec eval ast = match ast with
Int(x) -> x
|Add(x,y) -> let left = eval x in
              let right = eval y in
              left + right
...
```

TA-DA! We now have a way to get a single value from a string. Namely:

```
eval (parse (lex "+ 2 3")) = 5
```

Typically an interpreter will take a program statement by statement and do this exact thing. A compiler will look at a list of strings (read: text file) and compute an analogous text file.

Notice that technically, the meaning from this evaluator is created purely from what we decided Add would do. Additionally this language is very simple. Once we start adding more to our language, it may be possible that we have something that follows our grammar but makes no sense. Typically we can fix this by changing our parser, or rewriting the grammar (there are multiple different grammars that express the same language). In some cases, type checking is done during evaluation (like in ruby).

Suppose that I had a statement that looked like `"x+ 1"`. In most programming languages this is grammatically correct, variable added to a constant. However if $x = true$ then trying to do this operation would fail and it would be meaningless. Some languages get around this by casting, or by creating a new data type or behavior (see javascript and C).

This idea of defining behavior of a language is a branch of semantics. One particular type called operational semantics is the next topic.

Chapter 11

Operational Semantics

I am not a fully operational person

Cliff

11.1 Introduction

Now that we can design a language, we may want to do a few of things, two of which we will talking about here:

- Give meaning to the language
- Prove correctness of a program

Both of these goals can be achieved through the use of operational semantics. Semantics referring to the meaning of a statement, and operational referring to how something operates.

11.2 Meaning

If you ever take a philosophical linguistics course[1] you talk about some weird things that happen in languages, but you also talk about how meaning is sometimes attached to words. Slang in particular falls in and out of favor so figuring out how we attach additional meaning to words is always brought up. How would you define "vibe" to a non-native speaker when saying something like "Did not pass the vibe check"? How would you describe "mid" in a sentence like "Cliff was pretty mid last semester"? Operational semantics is a way to help describe the meaning of a statement in a programming language. Analogously, how do you describe the sentence `'fun x -> x 3'` to someone unfamiliar with functional programming?

There's plenty of ways that you can describe meaning. In programming language theory there are typically three major ways: denotations semantics, axiomatic semantics and operational semantics.

- Denotations: describe meaning via mathematical constructs
- Operational: describe meaning via how something operates
- Axiomatic: describing meanings via axioms

How I think about (and I am sure that people more in both the linguistics and PL space would be mad at me) is that denotational semantics is by giving a definition. For example: '"Blue" refers to light waves that fall in-between 450 and 495 nm'. Axiomatic semantics gives examples. For example: 'the sky, the ocean, and that person's eyes are blue'. Operational semantics describe how we use it. Example: '"Blue" is referring to a shade people see between green and violet'.

So when we talk about the meaning of a program, we want to talk about it in terms of how the program operates. More specifically, we use operational semantics to communicate language design ideas. If we want to talk about another language

[1] Would recommend Phil360: Philosophy of Language with Alexander Williams

however let's use some terms to help us. If I want to talk about some language x, then I will refer to x as the target language. The language that I will be describing x in, I will call the Meta-language. So If I want to talk about OCaml, then OCaml will be the target language, and English will be the Meta-language.

11.3 Correctness

When we talk about correctness, we basically mean, does the program run how we expect it to run? Can I prove that `+ 2 3` returns 5 in `Math-ew`? How can I prove that `(+ 2 (* 3 4))` returns 14 in LISP? The answer to this is not much different than proving that $((p \wedge q) \wedge (p \Rightarrow r) \wedge (q \Rightarrow r)) \rightarrow r$. Namely, we can make a proof:

$$\begin{array}{ll} & p \wedge q \\ & p \Rightarrow r \\ & q \Rightarrow r \\ \hline \therefore & r \end{array}$$

That is, if we know rules of things, we can derive new things. Suppose that we know that $3 * 4 = 12$ and we know that $12+2 = 14$. If we know these rules that we can say something like $2+3*4 = 14$ or `(+ 2 ( * 3 4))` returns 14. However instead of using defined rules of algebra or logic that we know, we are going to use defined rules of the target language.

11.4 Operational Semantics

Let us define a very basic language $Alanguge$:

$$\begin{array}{lll} e \rightarrow & n \\ \rightarrow & e \; ? \; e \\ n \rightarrow & 0|1|2|3|... \end{array}$$

A has really two statements that exist in the language. Let us make a rule that describes what we should do when met with either of these two statements.

If the statement in A is just n, then I want to evaluate to myself. So 3 should evaluate to 3, and 5 should evaluate to 5. This rule is pretty basic and so we could say this is an axiom in our language, or that we don't need proof to say that 3 is 3. So we use the following notation:

$$\frac{}{n \Rightarrow n}$$

This is just a conclusion, or something that is true in and of itself.

On the other hand, if I am given a statement that looks like 3?4 I want something to be evaluated to a value. In A, I want to use ? to add its two operands. So I may need to have a rule that describes my two operands, and what I should do when I see something that looks like e ? e.

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{e_1 \; ? \; e_2 \Rightarrow n_3}$$

This is to say that e_1 and e_2 are some expressions, and that e_1 will eventually evaluate to some number, n_1, while e_2 will eventually evaluate to some number n_2. We then need to describe that we want to add the two numbers together to get a final value: n_3 is $n_1 + n_2$. This part is described in our meta language. We then want to show that if these statements are true, then when we see e_1 ? e_2 that we want to return n_3, whatever that is.

This particular example that uses ? instead of +, is just to show you that we can just arbitrarily use symbols to stand for symbols and as long as we describe what this symbol does in our target language, then we can show you a rule of what is supposed to happen.

Now that we have our two rules to match each thing in our grammar, we can start making proofs that show what would happen if we had a statement like 4 ? 3.

In this example, looking at our grammar, we can see that 4 is e_1 and that 3 is e_2. We also know that, 4 and 3 are just numbers which we know evaluate to themselves. So constructing a proof of correctness for the statement 4 ? 4 using the

above two rules would look like:

$$\frac{\overline{4 \Rightarrow 4} \qquad \overline{3 \Rightarrow 3} \qquad 7 \text{ is } 4+3}{4 \,?\, 3 \Rightarrow 7}$$

That also means that we could prove larger expressions such as 3?4?5. Here I will assign 3 to e_1 and 4?5 to e_2, but you could instead say that 3?4 is e_1 and 5 is e_2. For this particular rule it does not matter, but depending on the operation, you may need to give more information so you don't get this ambiguous parse. The proof is as follows:

$$\frac{\dfrac{\overline{4 \Rightarrow 4} \qquad \overline{5 \Rightarrow 5} \qquad 9 \text{ is } 4+5}{4 \,?\, 5 \Rightarrow 9} \qquad \overline{3 \Rightarrow 3} \qquad 12 \text{ is } 3+9}{3 \,?\, 4 \,?\, 5 \Rightarrow 12}$$

As we add more to our language, we need to add more rules to our operational semantics. Let us consider the language *Blanguage*:

$$\begin{array}{rcl} e & \rightarrow & n \\ & \rightarrow & e \;+\; e \\ & \rightarrow & V \\ & \rightarrow & \text{let } V = e \text{ in } e \\ n & \rightarrow & 0|1|2|3|\ldots \\ V & \Rightarrow & a|b|c|d|\ldots \end{array}$$

I have changed the ? symbol to a + since we know that we just want to add the sub-expressions anyway. Additionally, we have now added variables to our language. By adding variables, we need to add something to our operational semantics: an environment.

Simply put, an environment is a mapping from variables to values.An example environment could be something like `[x:3,y:4]` We will denote an arbitrary environment with the character A. We will also need to update our rules to incorporate this environment. Let's first update our rules. The updated number and + rule are:

$$\frac{}{A; n \Rightarrow n}$$

$$\frac{A; e_1 \Rightarrow n_1 \qquad A; e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A; e_1 \;+\; e_2 \Rightarrow n_3}$$

What this means is that each expression e_x is being evaluated with the environment A. So suppose that we have previously bound the variable x is to the value 4. If we want to evaluate the statement 6 with this environment, then the proof would look like:

$$\frac{}{A, x : 4; 6 \Rightarrow 6}$$

I still include A because there are probably other environment variables that we are unaware of.

However, this is quite a boring example. What we may care about is how to look up a variable in our language. That is, what is the rule for $e \rightarrow V$? If we want to evaluate V into a value, we need to look up that value in the environment. Thus, our rule has to describe this process. Conventionally, we do this in the following way:

$$\frac{A(x) \Rightarrow v}{A; x \Rightarrow v}$$

So if had previously bound the variable x to the value 4 and wanted to look up x, it would look like:

$$\frac{A, x : 4; (x) \Rightarrow 4}{A, x : 4; x \Rightarrow 4}$$

This rule looks like it's just a repetition of a line, but recall the idea of the target language and the meta language. The conclusion is describing the target language, while the premise or hypothesis is describing what to do in the meta language.

We did do this a bit out of order. Before we can look up anything, we would have first needed to bind something. So let's describe the rule of $e \rightarrow \text{let } V = e_1 \text{ in } e_2$.

In this case, following OCaml (this is not always the case), before we bind a value to a variable, we want to evaluate the expression e_1 to a value and then bind that resulting value to the variable. Then we want to use this new binding when we are evaluating the expression e_2. Consider `let x = 3 in x + 1`. x+1 is the body and the binding we just made `x = 3` should be used when evaluating this. The rule that describes all this is the following:

$$\frac{A; e_1 \Rightarrow v \qquad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow e_3}$$

So in this case, we are evaluating e_1 to a value v, and then adding this binding to the environment when we evaluate e_2.

Using these rules, let us show a proof of correctness that let $x = 3$ in $x + 4$.

$$\frac{A; e_1 \Rightarrow v \qquad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

In this case we identify that 3 is e_1 and $x + 4$ is e_2.

$$\frac{A; 3 \Rightarrow v \qquad A, x : v; x + 4 \Rightarrow e_3}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We know that $\frac{}{3 \Rightarrow 3}$ so

$$\frac{\dfrac{}{A; 3 \Rightarrow 3} \qquad A, x : 3; x + 4 \Rightarrow e_3}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We then want to use our plus rule when evaluating $x + 4$

$$\frac{\dfrac{}{A; 3 \Rightarrow 3} \qquad \dfrac{A, x : 3; e_4 \rightarrow n_1 \qquad A, x : 3; e_5 \rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A, x : 3; x + 4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Here we can do what we did above an notice that in $x + 4$ that x is e_4 and 4 is e_5.

$$\frac{\dfrac{}{A; 3 \Rightarrow 3} \qquad \dfrac{A, x : 3; x \rightarrow n_1 \qquad A, x : 3; 4 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A, x : 3; x + 4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Based on our variable lookup rule we can say that $x \Rightarrow 3$ making $n_1 = 3$:

$$\frac{\dfrac{}{A; 3 \Rightarrow 3} \qquad \dfrac{\dfrac{A, x : 3; (x) \Rightarrow 3}{A, x : 3; x \rightarrow 3} \qquad A, x : 3; 4 \rightarrow n_2 \qquad n_3 \text{ is } 3 + n_2}{A, x : 3; x + 4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We know that 4 evaluates to itself:

$$\frac{\dfrac{}{A; 3 \Rightarrow 3} \qquad \dfrac{\dfrac{A, x : 3; (x) \Rightarrow 3}{A, x : 3; x \rightarrow 3} \qquad \dfrac{}{A, x : 3; 4 \rightarrow 4} \qquad n_3 \text{ is } 3 + 4}{A, x : 3; x + 4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

and we know that $3 + 4$ is the value of 7:

$$\frac{\dfrac{}{A; 3 \Rightarrow 3} \qquad \dfrac{\dfrac{A, x : 3; (x) \Rightarrow 3}{A, x : 3; x \rightarrow 3} \qquad \dfrac{}{A, x : 3; 4 \rightarrow 4} \qquad 7 \text{ is } 3 + 4}{A, x : 3; x + 4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Thus we know that e_3 is the final value of the expression.

$$\dfrac{\dfrac{}{A; 3 \Rightarrow 3} \qquad \dfrac{\dfrac{\dfrac{A, x : 3; (x) \Rightarrow 3}{A, x : 3; x \rightarrow 3} \qquad \dfrac{}{A, x : 3; 4 \rightarrow 4} \qquad 7 \text{ is } 3 + 4}{A, x : 3; x + 4 \Rightarrow 7}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow 7}$$

One point of confusion is what happens when we have a statement like `let x = 3 in let x = 4 in x + 5`. We would eventually get to a point where we have to evaluate $A, x : 3, x : 4; x$. The question of which x you use is dependent on the the rules you have. In this case, I am adding any new binding to the end of A ($A, x : v; e$) which means in order to get the scope correct, I need to choose the right most binding in the list.

Now that we have some more rules, we can keep adding things to to our grammar (to our language) and write more rules for how they should act.

Let us consider the language $Clanguage$ (not to be confused with C):

$$\begin{array}{rl} e \rightarrow & n \\ \rightarrow & e + e \\ \rightarrow & V \\ \rightarrow & \text{let } V = e \text{ in } e \\ \rightarrow & B \\ \rightarrow & \text{if } e \text{ then } e \text{ else } e \\ n \rightarrow & 0|1|2|3|\ldots \\ V \Rightarrow & a|b|c|d|\ldots \\ B \Rightarrow & true|false \end{array}$$

We should add some rules to incorporate these new values.

Let's start with our basic values of `true` and `false`.

$$\dfrac{}{A; true \Rightarrow true} \qquad \dfrac{}{A; false \Rightarrow false}$$

Moving onto our `if` rule, we have a few things that we could do. Here is one way to describe what to do:

$$\dfrac{A; e_1 \Rightarrow v_1 \qquad A; e_2 \Rightarrow v_2 \qquad A; e_3 \Rightarrow v_3 \qquad v_4 \text{ is } v_2 \text{ if } v_1 \text{ else } v_3}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_4}$$

This doesn't really tell us much and it tells us to evaluate e_3 even if $e1$ is `true`. This doesn't seem correct since we don't want to run code found in an `else` block if the guard is true. Perhaps we need to be more verbose about it:

$$\dfrac{A; e_1 \Rightarrow v_1 \qquad A; e_2 \Rightarrow v_2 \qquad v_1 == true}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_2}$$

This rule tells us to return v_2 if the hypothesis $v_1 == true$ is valid. See this example:

$$\dfrac{\dfrac{}{A; true \Rightarrow true} \qquad \dfrac{}{A; 3 \Rightarrow 3} \qquad true == true}{A; \text{if } true \text{ then } 3 \text{ else } 4 \Rightarrow 3}$$

If we had tried to prove something like the following:

$$\dfrac{\dfrac{}{A; false \Rightarrow false} \qquad \dfrac{}{A; 3 \Rightarrow 3} \qquad false == true}{A; \text{if } false \text{ then } 3 \text{ else } 4 \Rightarrow 3}$$

Notice that we get an invalid proof: `false == true` is logically incorrect and thus we couldn't make a valid proof. If we can't construct a valid proof, then we can say the program will not run how we claim it will run. Thus, we need to also add a rule about when the guard is false:

$$\dfrac{A; e_1 \Rightarrow v_1 \qquad A; e_3 \Rightarrow v_2 \qquad v_1 == false}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_2}$$

This means that we may need multiple rules to describe the behavior or meaning of one sentence. To describe the meaning of the if e_1 then e_2 else e_3 sentence, we needed the two rules:

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_1 == true}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_2} \qquad \frac{A; e_1 \Rightarrow v_1 \quad A; e_3 \Rightarrow v_2 \quad v_1 == false}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_2}$$

Altogether, the rules for $CIanguage$ is:

$$\frac{}{A; n \Rightarrow n} \qquad \frac{}{A; true \Rightarrow true} \qquad \frac{}{A; false \Rightarrow false}$$

$$\frac{A(x) \Rightarrow v}{A; x \Rightarrow v} \qquad \frac{A; e_1 \Rightarrow v \quad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow e_3} \qquad \frac{A; e_1 \Rightarrow n_1 \quad A; e_2 \Rightarrow n_2 \quad n_3 \text{ is } n_1 + n_2}{A; e_1 \ + \ e_2 \Rightarrow n_3}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_1 == true}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_2} \qquad \frac{A; e_1 \Rightarrow v_1 \quad A; e_3 \Rightarrow v_2 \quad v_1 == false}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_2}$$

11.5 Definitions interpreter

Let us go back to our very simple $Alanguage$. Recall the idea of Operational Semantics is to give meaning through how expressions operate. This is the basis of the idea of an interpreter. How a statement should evaluate is what the interpreter does. Thus, we can easily make an interpreter that is analogous to the operational semantics of a language.

Consider the rules of $Alangauge$

$$\frac{}{A; n \Rightarrow n}$$

$$\frac{A; e_1 \Rightarrow n_1 \quad A; e_2 \Rightarrow n_2 \quad n_3 \text{ is } n_1 + n_2}{A; e_1 \ + \ e_2 \Rightarrow n_3}$$

If we consider the premises and the final result of each rule, we can model an interpreter to do exactly what we specify in the rules.

Assuming we have a lexer and parser implemented and a type for both Numbers and an Add construct, we can write an interpreter that follows the above rules:

```
1 def rec eval expr env= match expr with
2  Num(x) -> x
3 |Add(e1,e2) -> let n1 = eval e1 env in
4                let n2 = eval e2 env in
5                let n3 = n1 + n2 in
6                n3;;
```

In moving to $BLanguage$, we gain more rules:

$$\frac{A(x) \Rightarrow v}{A; x \Rightarrow v}$$

$$\frac{A; e_1 \Rightarrow v \quad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow e_3}$$

We can then update our interpreter accordingly:

```
1 def rec eval expr env = match expr with
2  Num(x) -> x
3 |Add(e1,e2) ->   let n1 = eval e1 env in
4                  let n2 = eval e2 env in
5                  let n3 = n1 + n2 in
6                  n3
7 |Var(x) ->       let v = lookup env x in
8                  v
```

```
|Let(x,e1,e2) -> let v = eval e1 env in
                 let env' = update env (x,v) in
                 let e3 = eval e2 env' in
                 e3;;
```

This assumes that we have an function that adds variable and value pairs to the environment and a function that looks up a variable in the environment.

```
def rec eval expr env = match expr with
 Num(x) -> x
|Bool(x) -> x
|Add(e1,e2) ->   let n1 = eval e1 env in
                 let n2 = eval e2 env in
                 let n3 = n1 + n2
                 n3
|Var(x) ->       let v = lookup env x in
                 v
|Let(x,e1,e2) -> let v = eval e1 env in
                 let env' = update env (x,v) in
                 let e3 = eval e2 env' in
                 e3;;
|If(e1,e2,e3)
```

Chapter 12

Lambda Calculus

Baaaa

Sheep

12.1 Intro

According to Wikipedia[1], "*Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.*"

While correct, we will be treating it as a Turing complete language which is the basis of many functional languages. So we will go over how to create statements in this language, and how to evaluate them.

Let us begin by giving the grammar for the language:

$$\begin{aligned} e \Rightarrow \quad & x \\ & |\lambda x.e \\ & |ee \end{aligned}$$

x here is a variable. That's it. A very small language grammar. Using this grammar the following statements are valid:

x	$x\ y$	$\lambda x.x$	$\lambda x.y$
xyz	xx	$\lambda x.\lambda y.x$	$\lambda x.\lambda y.xy$

12.2 Turing Complete

As we saw earlier, finite state machine can only solve certain types of problems. Different machines have different levels of computational power. Checking if a string is accepted by a regular expression requires a Finite State Machine. Checking if a string is accepted by a Context Free Languages requires a Push Down Automata (PDA)[2]. Recursively enumerable languages need a Turing machine. A Turing Machine[3], can solve any computable problem. Some problems, we know cannot be solved: like the halting problem, thus a Turing machine cannot solve this problem. But if we know the problem can be solved, then a Turing machine, can model it.

It is important to note the difference of syntax and semantics here. CFGs and regular expressions only describe the syntax of the language. Since they only describe sets of strings, they say nothing about what those strings can express (in this case, a language is just a set of strings that is allowed). What a language is capable of expressing (or its semantics) is a different question (in this case, a language is a way to express ideas). We will now pull away from the former way of speaking about languages, and start discussing the latter. One property the semantics of a language could have is Turing

[1] https://en.wikipedia.org/wiki/Lambda_calculus

[2] We will not be covering PDAs in this course

[3] initially called a-machines or atomic machines

Completeness.

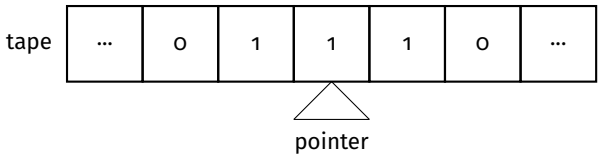
When we say a language is Turing complete, we mean that language can model or simulate a Turing machine.. A Universal Turing Machines (UTM), is a Turing machines that can produce other Turing machines, where each machine solves a particular problem (a machine that ANDs can't be used to OR). What does a Turing complete language look like? Well, despite the fact that the syntax of Lambda Calculus can be represented as a CFG, the semantics of the language is enough to be Turing complete.

12.3 Turing Machines

We said that a Turing complete language is one which can simulate a Turing machine. What exactly is a Turing Machine? It is a machine that has the following properties:

- Has an infinite ticker tape (a tape with an infinite number of cells)
- Each cell on the ticker tape stores a symbol from a finite alphabet (typically either a 1 or a 0)
- Has some "pointer" that can points to a cell on the tape
- The pointer needs to be able to move left or right one cell
- Has a writer and reader on the pointer to read the value in the cell or overwrite its value
- has a list of "states" that tell the pointer's reader and writer what to do and what other state to move to

Here is an example of a Turing Machine that will zero out the tape until a 0 is read in on both sides of the starting point:

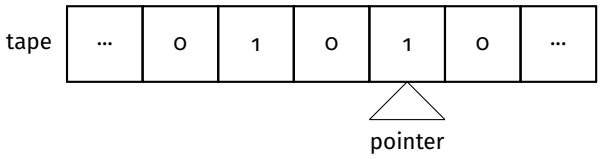


Read in	State A			State B		
	Write	Move	New State	Write	Move	New State
0	0	L	B	0	R	Halt
1	0	L	A	1	R	A

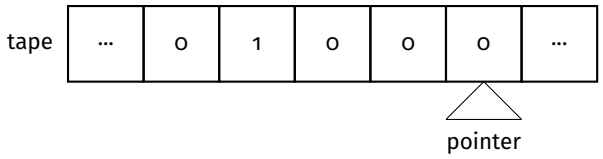
Assuming we start in the cell seen above and start in State A, here is the next few things that will happen:

1. We read in a 1, and are in state A so we write a 0 and move left, staying in state A
2. We read in another 1, and are in state A so we write a 0 and move left, staying in state A
3. We read in 0, and are in state B so we write a 0 and move Right and Halt.

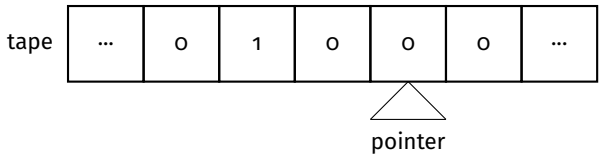
These steps can be visualized here:
After Step 1:



After Step 2:



After Step 3:



This process gets very complicated when writing larger programs, but this machine can be used to implement any computer algorithm. Thus any language that can simulate this machine (much like we simulated a FSM as a project in the course), then we say that language is Turing complete. Lambda Calculus is one such language that is Turing Complete.

12.4 Lambda Calculus Semantics

As we saw in the intro section, the syntax of the language is very small. It's now time to figure out what do those lambda calculus sentences mean.

12.4.1 Variables

Let's start with the basic `'x'` sentence. In languages like C, a variable by itself means whatever value is bound to that variable.

```
int x = 3;
x; //here x means 3
```

In lambda calculus, this meaning stays the same, however what the variable is bound too is sometimes not given.

```
  y //y not defined, but understood to be a variable
/*
Analogous to the following C code
void foo(){
  y; //not defined in this scope, but understood to be defined elsewhere.
}
*/
```

12.4.2 Function Definitions

The next type of sentence we is referred to as a lambda function. A lambda function takes the form $\lambda x.e$, where x is a variable and e is another lambda expression. As seen in the intro section, $\lambda y.x$ is a valid lambda expression, where y is a variable and x is another lambda expressions (in this case, a variable). This is called a function because it has the same semantics of a function we may see in other languages. Let's first break down it's structure though.

$$\lambda \boxed{\text{y}}.\underline{x}$$

This is a lambda function where the $\boxed{\text{y}}$ can be thought of as the parameter to the function, and $\underline{x}$ can be thought of as the body.
Since lambda functions are also lambda expressions, they can be placed as the body of another function: $\lambda x.\lambda y.y$. This one is more complicated, so let's look at it's structure:

$$\lambda \boxed{\text{x}}.\underline{\lambda \boxed{\text{y}}.y}$$

In this case, the λx function has parameter 'x' and body of $\lambda y.y$. The λy function has parameter 'y' and body 'y'.

Here is something analogous in other languages:

For reference: lambda calculus: $\lambda x.\lambda y.y$

```
# Python
lambda x: lambda y: y
```

```
(* Ocaml *)
fun x -> fun y -> y
```

```
# Ruby
Proc.new{|x| Proc.new{|y| y}}
```

```
// C
void* bar(void* y) { return y;}
void* (*foo(void* x))(void*){ return bar; }
```

```
// Java
interface Lambda{ Object run(int i); }
public static Lambda foo(Object x){
  return (Lambda)((y) -> y);
}
```

Each thing here is a function, that takes in one parameter, and then returns a function (where that returned function is the identity function).

As you can imagine, we can nest these even more: $\lambda x.\lambda y.\lambda z.z$. This becomes important when we change the body of the nested function: $\lambda x.\lambda y.x$ and start to do function application. More on this in a few sections.

12.4.3 Function Application

As the name suggests the last part is a way to call a function. However, this is also a bit of a misnomer, since sometimes we cannot apply a function. Let's expand on this.

$$(\lambda x.x)a$$

This is a lambda expression that consists of 2 sub-expressions: $(\lambda x.x)$ and a. When we have this structure, we call the left sub-expression using the right sub-expression as its input. Specifically, a will be used in the $(\lambda x.x)$ function. Since this function is the identity function, we just get back a.

$$(\lambda x.x)a \Rightarrow a$$

This is equivalent to calling a function and getting the return value. Here are some analogous examples in some programming languages:

For reference: lambda calculus: $(\lambda x.x)a$

```
# Python
(lambda x: x)(a)
```

```
(* Ocaml *)
(fun x -> x) a
```

```
# Ruby
Proc.new{|x| x}.call(a)
```

```
// C
void* foo(void* y) { return y;}
foo(a);
```

```
// Java
interface Lambda{ Object run(Object i); }
((Lambda)(x) -> x).run(a)
```

Notice, that we have an argument, and it is being replaced in the body with whatever the input is. Thus when given something like $(\lambda x.y)a$, we get y back. To see this in more detail, consider the following:

```
1 int foo(int x){
2   return x;
3 }
4 foo(3) //returns 3, because 3 is our input, where ever we see x, replace with 3.
5
6 int bar(int x){
7   return 4;
8 }
9 bar(3) //returns 4, because 3 is our input, but we don't use x anywhere
```

To see more complicated examples, we first need to consider the following lambda expression:

$$a\ b$$

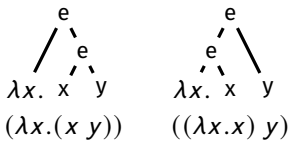
This lambda expression has two sub-expressions: a and b. Here however, a is not a lambda function, so we have no idea how to call a with b as input. In this case, since we cannot call a function, we say this expression means exactly what it says: $a\ b$.
To see this side by side:
$(\lambda x.x)\ a \Rightarrow a \mid (\lambda x.y)\ a \Rightarrow y \mid a\ b \Rightarrow a\ b$

12.5 Removing Ambiguity

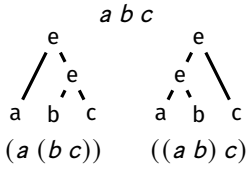
Now that we have the basics down, let's consider something more complicated:

$$\lambda x.x\ y$$

We can see this is an ambiguous statement. Either $x\ y$ the body of the λx function, or y is being applied to the $\lambda x.x$ function.



Here is another example of an ambiguous expression:



It is initially unclear if we are calling a with the input $b\ c$ or if we are calling a with input b and then calling c on the result of $a\ b$.
To help remove ambiguity, there are some explicit rules that Lambda Calculus follows.

- Expressions are left associative
- The scope of a function goes until the end of the entire expression or until a (unmatched) parenthesis is reached

12.5.1 Left Associative

When we say expressions are left associative, that means when given a series of expressions, for example three: $e_1\ e_2\ e_3$, then we group the left two items together before grouping the next item: $(e_1\ e_2)\ e_3$. This means that we take the right tree in the above example with $a\ b\ c$. As you can imagine, we chain this rule with larger expressions. Thus: $a\ b\ c\ d\ e\ f$ has implicit parenthesis: $(((((a\ b)\ c)\ d)\ e)\ f)$.

For functions, this is important since it tells us what order we should apply things:

$$\begin{aligned} & (\lambda x.\lambda y.y)\ a\ b \\ \Rightarrow\quad & ((\lambda x.\lambda y.y)\ a)\ b \\ \Rightarrow\quad & (\lambda y.y)\ b \\ \Rightarrow\quad & b \end{aligned}$$

12.5.2 Function Scope

The next important part is the scope of a lambda function. The body of of a lambda function is whatever follows the . symbol until the end of the entire expression is reached or a (unmatched) parenthesis is reached. It is important to note the parenthesis must be unmatched from the viewpoint of the lambda. Consider the following functions and their body (which is underlined):

$$\lambda x.\underline{a\ b\ y} \quad \lambda x.\underline{(a\ b)\ y} \quad \lambda x.\underline{a\ (b\ y)} \quad (\lambda x.\underline{(a\ b)})\ y \quad (\lambda x.\underline{(a\ b)\ y})\ z \quad (\lambda x.\underline{(a\ b)})\ y\ z$$

This rule continues even when we nest lambda functions:

$$\lambda x.\lambda y.\underline{a\ b\ y} \quad \lambda x.\lambda y.\underline{(a\ b)\ y} \quad \lambda x.\lambda y.\underline{a\ (b\ y)} \quad \lambda x.\underline{(\lambda y.\underline{a\ b})\ y} \quad (\lambda x.\underline{(\lambda y.\underline{a})\ b})\ y$$

12.6 Reduction

The process of of applying a function is called reducing. In particular, we call a function call a beta reduction (β-reduction). A single function call is a reduction. So the following is actually performing two reductions.

$$\begin{array}{lll} & (\lambda x.\lambda y.y)\ a\ b & \\ \Rightarrow & ((\lambda x.\lambda y.y)\ a)\ b & \\ \Rightarrow & (\lambda y.y)\ b & \text{reduced the x function} \\ \Rightarrow & b & \text{reduced the y function} \end{array}$$

When you cannot reduce any further, we say the expression is in beta normal form. So b is the result of reducing $(\lambda x.\lambda y.y)\ a\ b$ to beta normal form.

When performing a beta reduction, consider there is the function being called, and the expression passed in to the function. For example: $(\lambda x.\ x)\ y$ can be beta reduced by calling the $\lambda x.\ x$ function with the input y.

What happens when the input can also be reduced? $(\lambda x.\ a\ x\ a)\ ((\lambda y.\ y\ y)\ z)$ We have two options:

- We evaluate the argument first. That is, evaluate $((\lambda y.\ y\ y)\ z)$ to $(z\ z)$ and pass that into $(\lambda x.\ a\ x\ a)$
- We shove $((\lambda y.\ y\ y)\ z)$ into $(\lambda x.\ a\ x\ a)$ first and get $a\ ((\lambda y.\ y\ y)\ z)\ a$

The first is an example of **eager** evaluation (sometimes called call-by-value), where we evaluate the argument first before passing it into the function.

The second is an example of **lazy** evaluation (sometimes called call-by-name), where we only evaluate when we need to.

For example:

```
def foo(x):
  return "str"

foo(3+4)
# using lazy evaluation, 3 + 4 is never calculated
# using eager, 3 + 4 is calculated and then never used.
# in this case lazy does less work

# Try running print(foo((lambda x: print(x))(4))).
# Does python use eager or lazy evaluation?
```

12.7 Variable Semantics

Up until this point, we have been using simple functions, but typically more complicated or harder to read functions are used. For example:

$$(\lambda x.(\lambda x.(\lambda y.x\ y))\ x)\ x$$

To figure out how to reduce this problem, we need to discuss variables and their binding.

Variables fall under two categories: free or bound. A bound variable is one who's value is dependent on a parameter of a lambda function.

$$(\lambda x.\underline{x}\ \boxed{\text{a}})\ \boxed{\text{b}}$$

In the above example, x is bound to the input parameter since they share the same name and x is in the body of the λx function. a and b are then what are known as free variables, variables not dependent on the parameter. b is not bound because it falls outside the body of the function, and does not share the same name as the parameter. a is not bound because it does not share the same name as the parameter. Consider the following C program:

```c
int a = 6; // free
int foo(int x){ // x is name of the parameter
  x == 5; // this x is bound to the parameter
  int y = 3; // free
}
```

In regards to the `foo` function, a and y are free since they are not bound to the parameter x.

This becomes important when we nest functions:

$$(\lambda x.\lambda y.\underline{x}\ \boxed{\text{a}}\ \underline{y})\ \boxed{\text{b}}$$

Here, x is bound to the outer λx parameter and the y is bound to the inner λy parameter. This gets a tad confusing when we shadow the variable:

$$(\lambda x.\lambda x.\underline{x})\ \boxed{\text{a}}$$

In these cases, we know that x is bound, but to what? In lambda calculus (and most languages that I know), the x is bound to the inner λ function. This is because the parameter is being shadowed by the inner function. Consider the following C code:

```c
int a = 6; // free
{
  int a = 5;
  printf("%d\n",a); //prints 5 here since a is shadowed by the previous line
}
printf("%d\n",a); //prints 6, now that the inner 5 is out of scope
```

Thus, we can now beta reduce complicated functions if we keep this is mind:

$$\begin{aligned} & (\lambda x.(\lambda x.x\ x)\ x)\ a \\ \Rightarrow\ & (\lambda x.x\ x)\ a \\ \Rightarrow\ & a\ a \end{aligned}$$

Here, the right most x is bound to the left most λx where as the inner two x's are bound to the inner λ function.

$$(\lambda\underline{x}.(\lambda\boxed{\text{x}}.\boxed{\text{x}}\ \boxed{\text{x}})\ \underline{x})$$

Just reiterated, where all boxed variables are related and all underlined variables are related. Here's one more for practice:

$$\begin{aligned} & (\lambda y.(\lambda x.x\ x)\ y\ x)\ a \\ \Rightarrow\ & (\lambda x.x\ x)\ a\ x \\ \Rightarrow\ & (a\ a)\ x \end{aligned}$$

To help make things more readable, there is this concept of alpha equivalence (α-equivalence). Alpha equivalence should not change the meaning of the initial statement, but rather make sure things are more readable, or to preserve the semantics of the initial statement.

To do so, an alpha-conversion (α-conversion) is the process of renaming all the variables that are bound together along with the bounded parameter to a different name.

$$(\lambda x.x)a \Rightarrow (\lambda y.y)a$$

Again, this does not change the semantics of the expression, but makes things more readable. Consider the C code:

```
1 int foo(int x){
2   return x + 1;
3 }
4 int bar(int y){
5   return y + 1;
6 }
```

There is no difference between `foo` and `bar`. These two functions are α-equivalent. So let's consider our initial statement and alpha-convert it to be more readable:

$$(\lambda x.(\lambda x.(\lambda y.x\ y))\ x)\ x \Rightarrow (\lambda a.(\lambda b.(\lambda y.b\ y))\ a)\ x$$

It is important to note that you **cannot** convert free variables. You can only convert bound variables. Thus $(\lambda x.x)\ x$ is not alpha equivalent to $(\lambda x.x)\ a$.

For the most part, this just helps with readability, but sometimes it is important to alpha convert to keep the semantics. Consider the following **incorrect** reduction:

$$\begin{array}{ll} & (\lambda y.(\lambda x.y))\ x \\ \Rightarrow & (\lambda x.x) \end{array}$$

This now reduces to the identity function, but this is incorrect since the initial outer x was free and now becomes bound. To keep the semantics, free variables cannot become bound, and bound variables cannot become free. To make this correct, we must alpha convert:

$$\begin{array}{ll} & (\lambda y.(\lambda x.y))\ x \\ \Rightarrow & (\lambda y.(\lambda a.y))\ x \\ \Rightarrow & (\lambda a.x) \end{array}$$

12.8 Church Encodings

Now that we have an idea of how to evaluate Lambda Calculus, let's discuss about how we can use this as a language to calculate information.

The words we use to represent concepts are ultimately arbitrary. We all came together and agree that words like 'true' or 'false' mean something. However, these things are ultimately arbitrary when figuring out the string of symbols to represent these concepts. In OCaml for example, we say `true` and `false`, but in Python we say `True` and `False`. Even in C, there is no boolean, it's (typically) just checking if a number is 0 or not.

It then stands to reason that in lambda calculus, while we may not have strings like "true" or "false", we do have something that represents these ideas. This concept is called encoding. We want to encode information into a text string valid in the language.

The encodings that exist in Lambda Calc were made by Alonzo Church who also introduced the idea of Lambda Calculus. He encoded '`true`' and '`false`' as follows:

- True: $\lambda x.\lambda y.x$
- False: $\lambda x.\lambda y.y$

That is, the OCaml program: `true` is analogous to the lambda calculus program: $\lambda x.\lambda y.x$

This may make sense once we introduce the `if guard then true_branch else false_branch` equivalent. To write `if a then b else c` in Lambda Calculus, we just write: `a b c`. This may be confusing so consider:

- `if true then false else true`: $(\lambda x.\lambda y.x)(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)$
- `if false then false else false`: $(\lambda x.\lambda y.y)(\lambda x.\lambda y.y)(\lambda x.\lambda y.y)$
- `if false then true else false`: $(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)(\lambda x.\lambda y.y)$

To see these encodings at work, consider `if true then false else true` should evaluate to `false`. So let's see what `if true then false else true` looks like in lambda calc

$$\begin{aligned}\text{if true then false else true} &= (\lambda x.\lambda y.x)\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)\\ &\Rightarrow ((\lambda x.\lambda y.x)\ (\lambda x.\lambda y.y))\ (\lambda x.\lambda y.x)\\ &\Rightarrow (\lambda y.(\lambda x.\lambda y.y))\ (\lambda x.\lambda y.x)\\ &\Rightarrow (\lambda x.\lambda y.y))\\ &= \text{false}\end{aligned}$$

There are other encodings that exist as well, such as pairs, numbers, addition and multiplication of numbers, and,or,not on booleans, etc. See Appendix ... TODO for more examples.

12.9 Looping

One such properties of Turing Completeness is the ability to jump, conditionally, or unconditionally. This ultimately allows for looping to exist so let's examine looping in Lambda Calc.

Let us consider the following lambda calculus expression:

$$(\lambda x.xx)(\lambda x.xx)$$

If were to beta reduce this, we would get back the exact same thing. This is one such example of a lambda calculus expression which would loop to infinity and never reach a beta normal form. This particular expression is called the Ω-combinator. This expression by itself is not truly useful, but we can exploit this structure and insert a modification to achieve a conditional or at least "recursive" looping structure.

Since lambda calculus doesn't have named functions, we cannot get recursion in the same way would could in typical programming languages. However, as we saw in OCaml, functions are pieces of data and the same is true for lambda calculus. The trick here is to pass in the recursive function into a wrapper function. In lambda calculus, we will be using the Y-combinator (sometimes called a fixpoint combinator). It is as follows:

$$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$$

To see this, suppose we have a function **F**. A recursive call may look something like **F(F(F(F(...F(**`value`**...))))**. We can obtain that using the following

$$\begin{aligned}&(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ F\\ \Rightarrow\ &(\lambda x.F(xx))(\lambda x.F(xx))\\ \Rightarrow\ &(F((\lambda x.F(xx))(\lambda x.F(xx))))\\ \Rightarrow\ &(F(F((\lambda x.F(xx))(\lambda x.F(xx)))))\\ \Rightarrow\ &(F(F(F((\lambda x.F(xx))(\lambda x.F(xx))))))\\ \Rightarrow\ &\ldots\end{aligned}$$

If we wanted to make this easier to read, let Y = $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$ and Y F = $((\lambda x.F(xx))(\lambda x.F(xx)))$

$$\begin{aligned}Y\ F = &\ (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ F\\ \Rightarrow &\ (\lambda x.F(xx))(\lambda x.F(xx))\\ \Rightarrow &\ (F((\lambda x.F(xx))(\lambda x.F(xx))))\\ \Rightarrow &\ (F(Y\ F))\\ \Rightarrow &\ \ldots\end{aligned}$$

So now suppose we could write encode numbers in lambda calculus much like we could "true" and "false" (we can![4]). Also suppose we could encode things like "=0", "n*m" and "n-1" like we could "if then else" and "and" and "or" (we can![5]). This could mean we could write a function G like factorial in the form $G = \lambda f.(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (f(n-1)))$
Now we can use G and a factorial number to place into Y.

$$
\begin{aligned}
(Y\ G)3 = \quad & ((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)3 \\
\Rightarrow \quad & ((\lambda x.G(xx))(\lambda x.G(xx)))3 \\
\Rightarrow \quad & (G(\lambda x.G(xx))(\lambda x.G(xx)))3 \\
\Rightarrow \quad & (G(Y\ G))3 \\
\Rightarrow \quad & \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * ((YG)2) \\
\Rightarrow \quad & 3 * ((YG)2) \\
\Rightarrow \quad & 3 * (((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)2) \\
\Rightarrow \quad & 3 * (((\lambda x.G(xx))(\lambda x.G(xx)))2) \\
\Rightarrow \quad & 3 * ((G(\lambda x.G(xx))(\lambda x.G(xx)))2) \\
\Rightarrow \quad & 3 * ((G(Y\ G))2) \\
\Rightarrow \quad & 3 * (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * ((YG)1)) \\
\Rightarrow \quad & 3 * (2 * ((YG)1)) \\
\Rightarrow \quad & 3 * (2 * (((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)1)) \\
\Rightarrow \quad & 3 * (2 * (((\lambda x.G(xx))(\lambda x.G(xx)))1)) \\
\Rightarrow \quad & 3 * (2 * ((G(Y\ G))1)) \\
\Rightarrow \quad & 3 * (2 * \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * ((YG)0)) \\
\Rightarrow \quad & 3 * (2 * (1 * ((YG)0))) \\
\Rightarrow \quad & 3 * (2 * (1 * (((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)0))) \\
\Rightarrow \quad & 3 * (2 * (1 * (((\lambda x.G(xx))(\lambda x.G(xx)))9))) \\
\Rightarrow \quad & 3 * (2 * (1 * ((G(Y\ G))0))) \\
\Rightarrow \quad & 3 * (2 * (1 * \text{if } 0 = 0 \text{ then } 1 \text{ else } 1 * ((YG)0))) \\
\Rightarrow \quad & 3 * (2 * (1 * 1)) \\
\Rightarrow \quad & 3 * (2 * (1)) \\
\Rightarrow \quad & 3 * (2) \\
\Rightarrow \quad & 6
\end{aligned}
$$

While this does look gross, I would highly recommend you trace through this on your own.

[4] See Appendix ... TODO
[5] See Appendix ... TODO

Chapter 13

Garbage Collection

It's called a Garbage Can, not a Garbage Cannot

13.1 Introduction

When we are implementing a language, there may be times when we need more information than just what is given to us and our operational semantics. We saw this when we added variables to our language: we needed an environment to keep track of variables and what they were bound to. This environemnt acted as our memory, allowing us to make a variable and store a value with it. When we were done with the scope of the variable, then the binding was automatically dropped (thanks to the immutability of OCaml). This automatic memory management is analogous to the stack, where items are pushed and popped automatically (more on this later), as opposed to the heap, where either the user has to management their own memory (C) or a garbage collector will manage the allocation and deallocation (OCaml, Java, etc).

Since we will be good people, we will not push the burden of memory management to the user. We will make the garbage collector. To do so, we need to consider what makes a good garbage collector.

A good garbage collector must take a conservative approach to deallocating memory. Consider:

Memory	Don't free	Free
In use	Good	Really Bad
Not in use	Eh	Good

Notice that we want to free memory that is no longer in use, and not free memory that is in use. If we have memory that we don't use and do not free it, all that really happens is we lose useable memory. This could mean we run out of memory, or things take longer to lookup in memory (decreasing time performance). Ultimately, the program will not be *wrong* per se, rather it will just be unoptimized.

Alternatively, we could have a garbage collector that frees things that are in use. This can cause the program to fail and potentially have dangerous side effects[1].

Determining if something is in use or not to be freed is the goal of garbage collection. However, while some things are easy to figure out if they are in use or not, some things are more ambiguous. To be a conservative garbage collector means to err on the side of caution, and only free things that we are guaranteed to be no longer in use. So how do we know if something is in use or not?

We will see three basic ideas to determining the answer to this question. However, it is important to note that modern garbage collector typically will use a modified or combination of the following ideas.

[1] Technically not freeing things that need to can also lead to security vulnerabilities too

13.2 Reference Counting

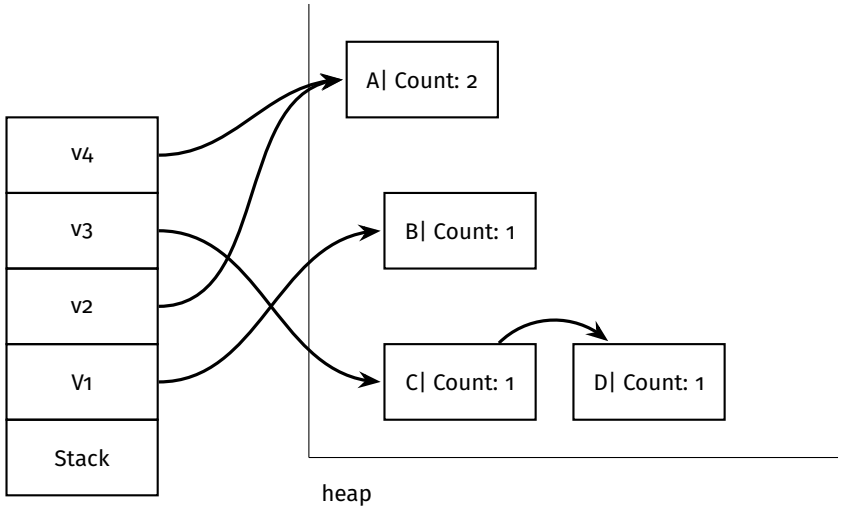
We say that a piece of memory is in use if we can reach that piece of data. If we loose a reference to a segment of memory, then we can't really use what's stored there so we can free it. One idea is to keep track of how many pointers (references) point to a segment of memory, and deallocate (free) when that counter reaches 0. Consider:

```
1 {
2   int* v1 = malloc(sizeof(int)); //say memory address 0xfff
3   // one thing points to 0xfff
4   {
5   int* v2 = v1; //now 2 things point to 0xfff
6   }
7   //now 1 thing points to 0xfff
8 }
9 //now nothing points to 0xfff
```

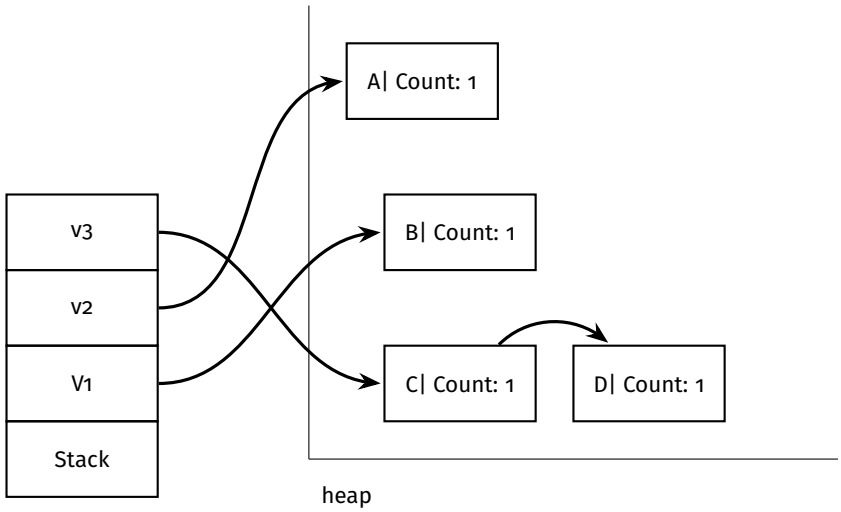
Recall that variables are valid until the end of their scope, and you can create a temporary scope with curly braces({}). Thus, `v1` is valid until line 6, while `v2` is valid until line 5. If we consider how many things are pointing to memory address `0xfff` at each line, we can see that the count increases when we allocate, or set a pointer to a variable. The count then decrements when the pointer goes out of scope.

It is important to note that the counter is incremented everytime a pointer to that segment of memory if updated (added or deleted). While this is a constant time operation, this typically ends up being called quite a lot of times. Additionally, space for the counter needs to be allocated with each item on the heap so there is some added space complexity to consider.

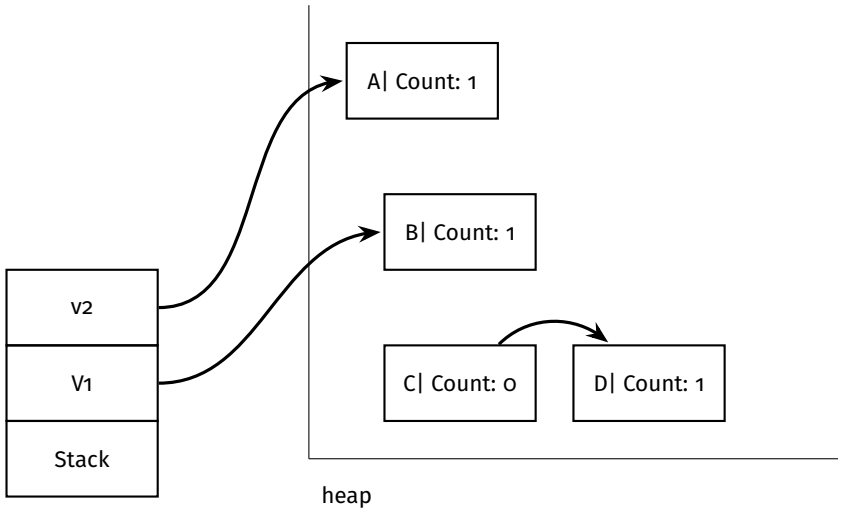
Consider the memory map:



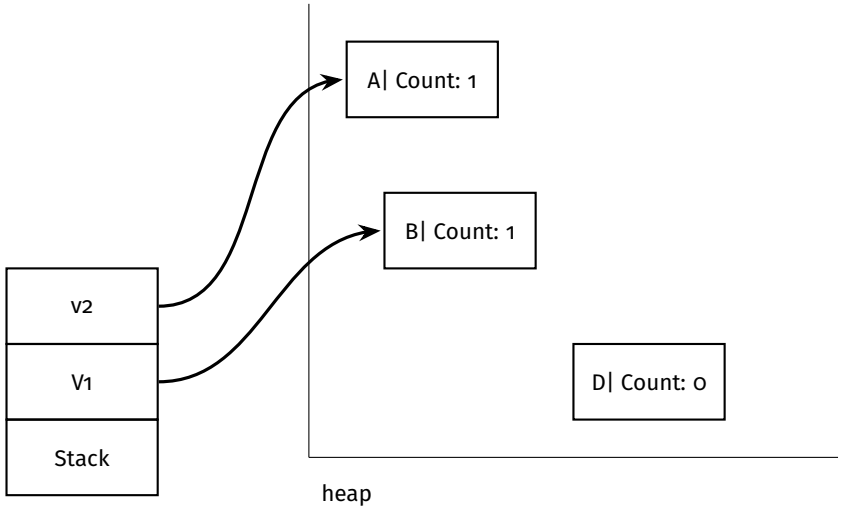
If v4 went out of scope, then the reference count of item A would be decremented to 2.



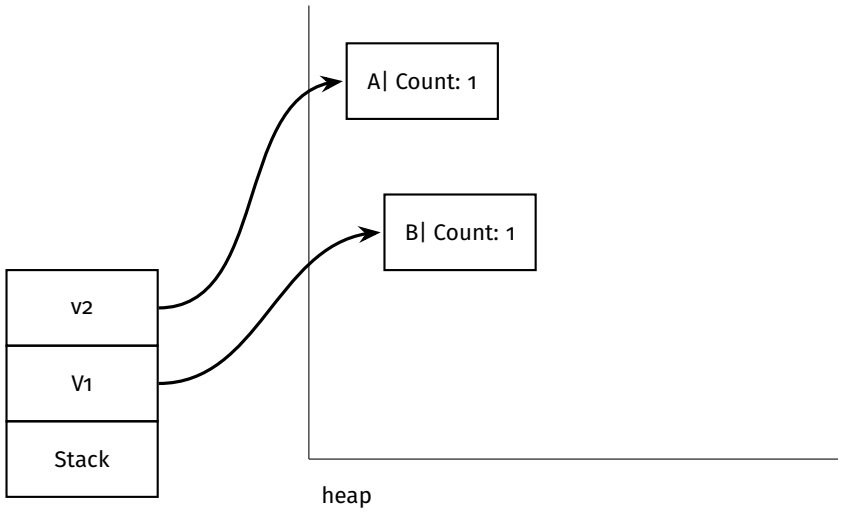
If v3 was popped off, then C would be at count 0.



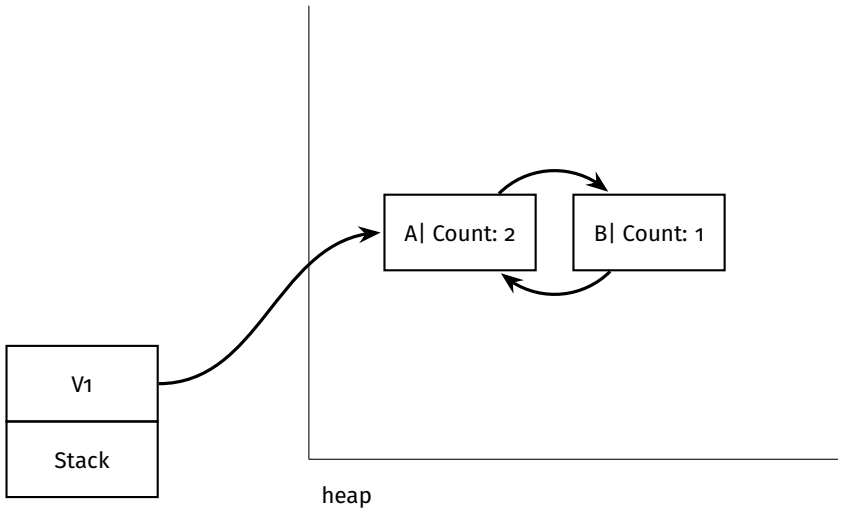
Since the count of C hits zero, then we free that item.



Now since the freeing of C caused the counter of D to decrement to zero, then D has to be freed as well.



One issue to consider is the idea of cyclic data. What happens when v1 is popped off the stack in the following memory diagram?



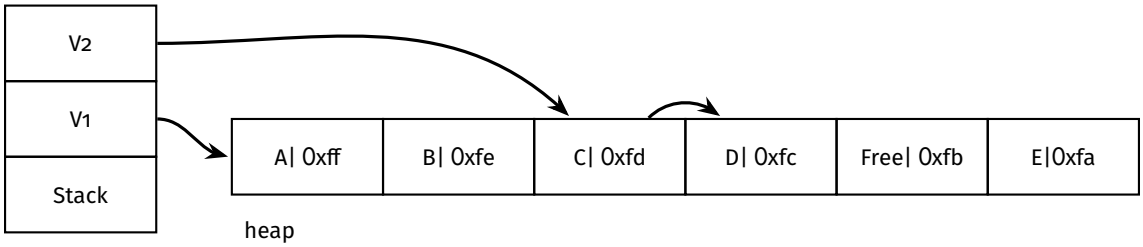
13.3 Mark and Sweep

Mark and Sweep is the next garbage collection strategy we will talk about, and it is a form of tracing garbage collection. There are various variations of Mark and Sweep and we will talk about one of the most basic forms.

In Mark and sweep we consider what is reachable based off what you can get to via the stack. However, we also need to go through and actually free everything that should be freed. In order to do so, we need to go through the entire heap, and figure out if what we are looking at is reachable from the stack or not[2]

The heap is just a linear piece of memory so let's restructure the picture of the heap:

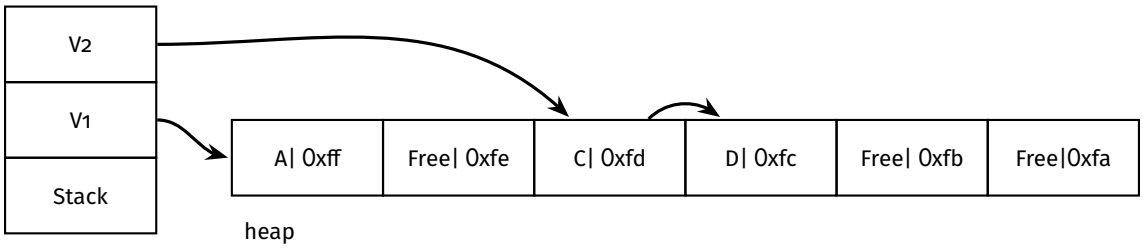
[2] some implementations have you go through stack then heap in O(n+m) fashion. Here we will be doing O(m*n).



In order to figure out what is in use and what is not in use, we can iterate through the entire heap, figure out if we can reach where we are looking via the stack.

- We look at `0xff` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfe` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfd` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfc` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfb` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfa` and we check if anything on the stack can reach it or it is already free.

After we do all of this, the only places of memory that fail this check are `0xfe` and `0xfa`. We then know we can free these two places in memory.



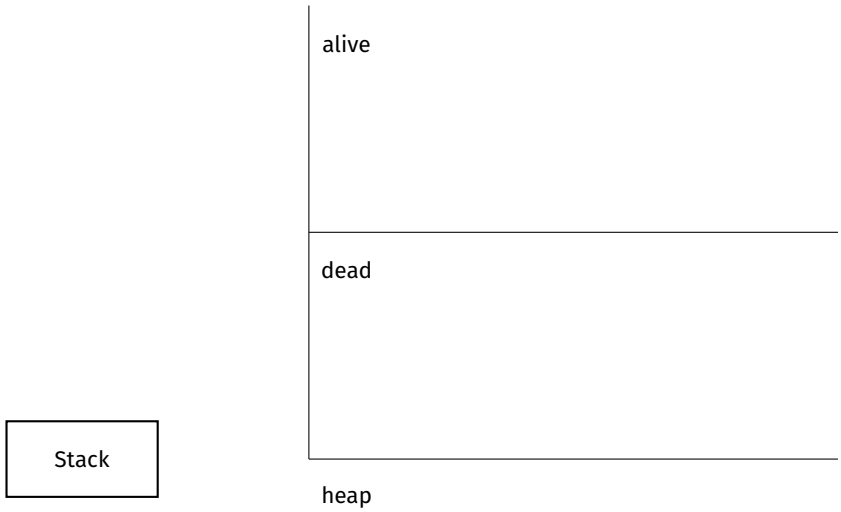
Note: `0xfc` is reachable from the stack, just not directly.

One thing to note is that the program must be paused while this is happening since we do not want things to be allocated or freed while this is occurring.

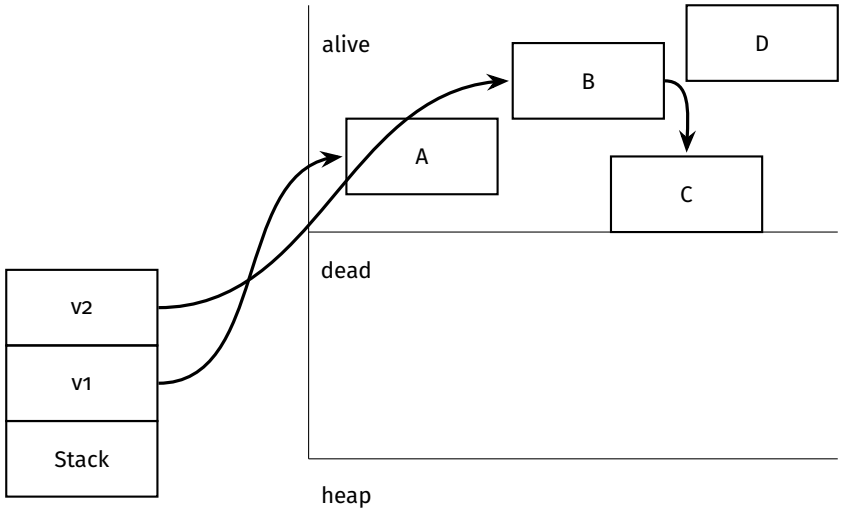
13.4 Stop and Copy

In the same vein of Mark and Sweep, have another tracing garb age collector: stop and Copy. Much like Mark and Sweep, the program must stop while this is occurring.

In Stop and Copy, we must first partition the Heap into an alive partition and a dead partition.

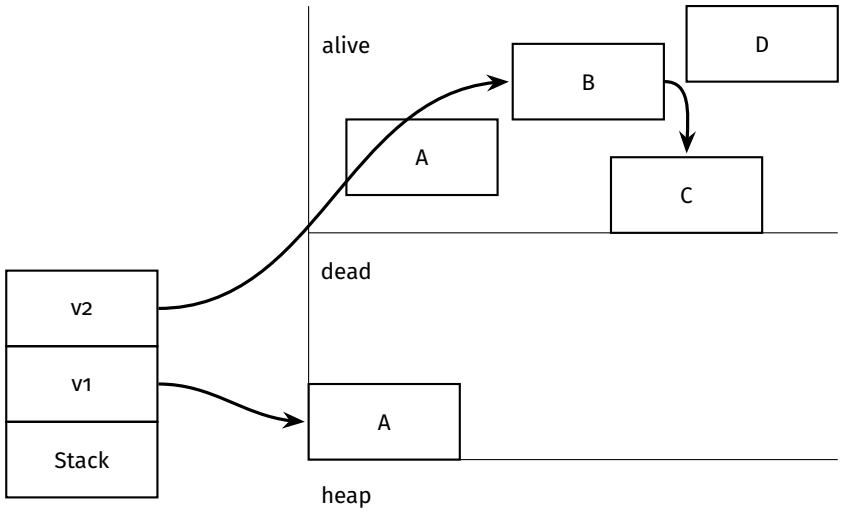


Then whenever we need to allocate something, we do so in the alive part.

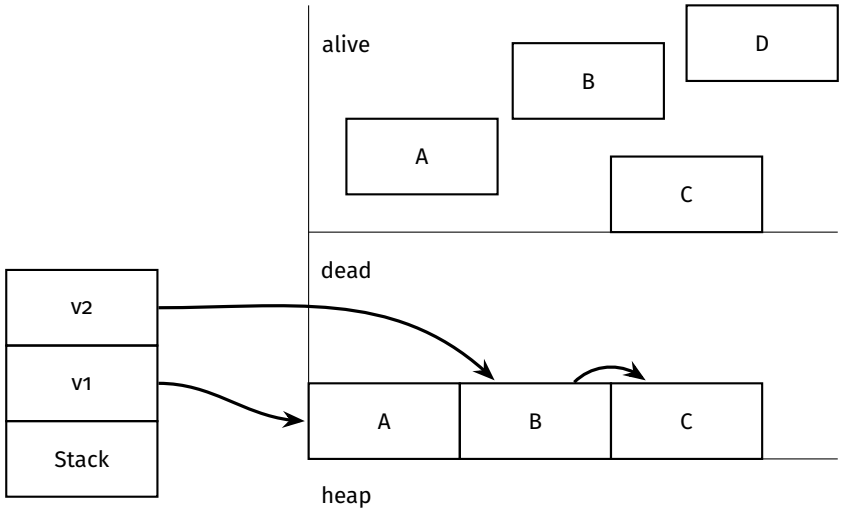


Then, when running garbage collection, we go through the entire stack, and copy over everything that is reachable to the dead partition.

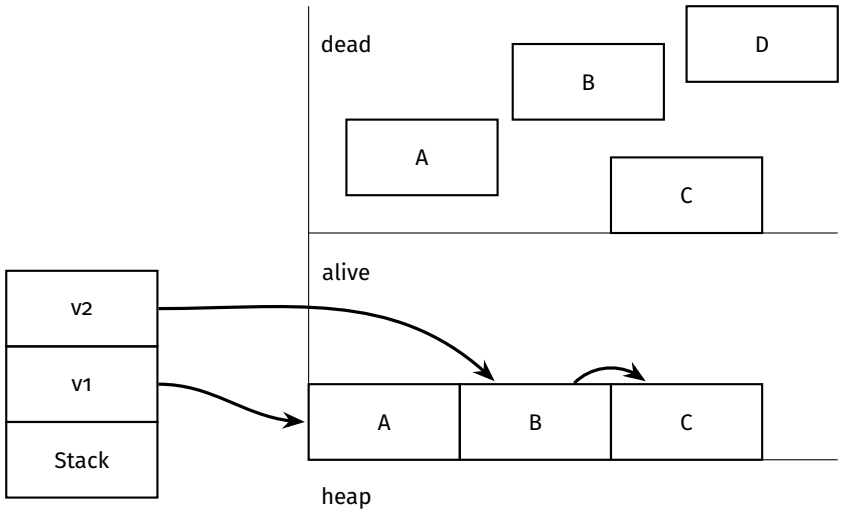
So first we look at say, v1 and copy whatever it points to to the dead partition.



We then do so for all other items on the stack:

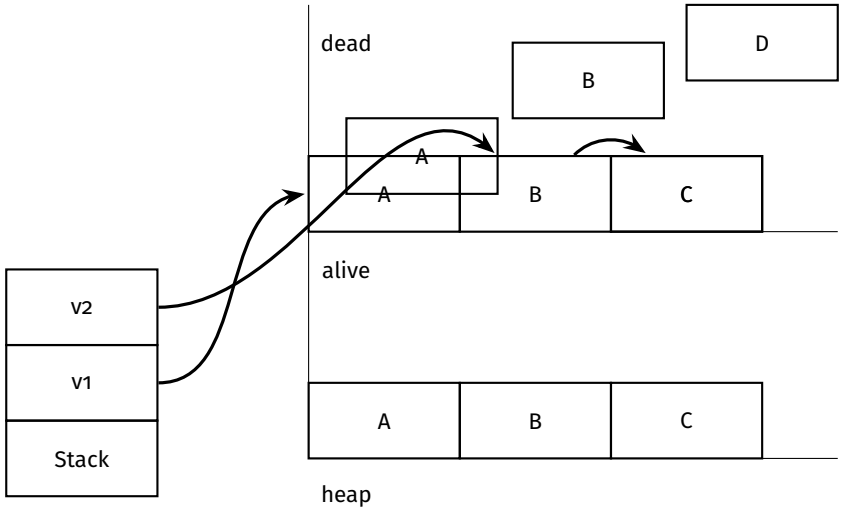


Then we swap partitions and continue.



We will continue to allocate in the alive section only, and when garbage collection happens again, we will copy every in-use memory over and swap partitions again. Due to the fact that this will copy over data, we can actually defragment our memory to optimize later memory allocation. This is important since we start off by halving our useable memory space to begin with.

Notice that when this happens again, we will just copy right over our past memory usage.



Chapter 14

Rust

Forgive me, I am a tad Rusty

Rustaceans

First, let me begin by saying that these notes are only a condensed and high level account of what is found in The Rust Book: https://doc.rust-lang.org/book/. I would definitely recommend you take a look there as nothing I could do would be as good. The topics covered per semester change rapidly depending on time and who is teaching. Historically we focus on what makes Rust unique (Ownership, Lifetimes and Smart pointers, chapters 4, 10, and 15 respectively).

This is a programming language chapter so it has two (2) main things: talk about some properties that the Rust programming language has and the syntax the language has. If you want to code along, all you need is a working version of Rust and a text editor. You can check to see if you have Rust installed by running `rustc -version`. At the time of writing, I am using Ruby 1.65.0.

Unlike other programming chapters, there will be a lot about the properties of the language itself, since Rust has a few new things not found in other languages. Before reading this chapter, please refer to the Garbage Collection Notes.

14.1 Introduction

Rust was made around 2016 from the folks over at Mozilla (the firefox[1] people). They built the first Rust Compiler in Ocaml, which means they really liked that language, so you may see some Ocaml-ness in the Rust Language. Rust's goal is to be a **safe** language, but at the same time, maintain the speed and find grain control of the machine that C gives you. Before we get into all that though, let's write our very first Rust program:

```
1 // hello_world.rs
2 fn main() {
3     println!("Hello, world!");
4 }
```

Despite this being very simple, we've already learned several things. We can also notice that we are going back to more "traditional" style languages.

- Single-line comments are started with the backslash
- Semicolons!
- `println!` is used to print things out to stdout
- functions use parenthesis (weird we need to say this)
- Rust file entension is `.rs`

[1] Firefox over Chrome-based browsers, always. Ad-blockers ftw

- Strings exist in the language (most languages have them, but some do not)
- Need a `main` function.

Can you think of more?

Now to compile our program we can just use

```
rustc hello_world.rs
./hello_world
```

Congrats, you have just made your first program in Rust! There are some things to note however:

- Rust is a compiled Language
- `rustc` is the rustc compiler (some compilers like to take the name of the language and add 'c' to the end: javac, ocamlc, rustc).
- If you run an `ls` you will notice that the executable `hello_world` was created. There were no other files made (like headers or object files).
- `rustc` is wrapped in a nice program called `cargo`. `Cargo` will allow you to create, compile, run, and test your Rust programs without much overhead. We use `cargo` to help manage your projects.

14.2 Memory and Security

Again, I would recommend that you go read the Garbage Collection Notes before this section.

C is considered a low level language because it doesn't really abstract things. We could think of C as an API for assembly. This makes C a very fast language because it doesn't have to deal much with garbage collection, or type checking during runtime. This also makes C a very unsafe language, because it is up to the programmer to manage memory and check types. Higher level languages abstract this away making for safer languages, but also makes the languages slower by comparison. Rust wants the best of both worlds here. It wants to be safe, but also be fast. For Rust to do this, it would need to forgo some of these costly overheads like garbage collection and type checking at runtime and instead drop them completely OR lean more on the compiler. Forgoing these options would just be C, so Rust is designed around how it's compiler can do a lot of work to get fast and safe programs.

14.2.1 Safety

What is safety? Consider the following:

- A server that sends out restricted information like bank passwords to anyone is unsafe.
- An air traffic control application that can be taken down via a DOS (denial of service) attack leading to plane crashes is unsafe.
- A grades server where you can overwrite everyone's grades with an F is unsafe.

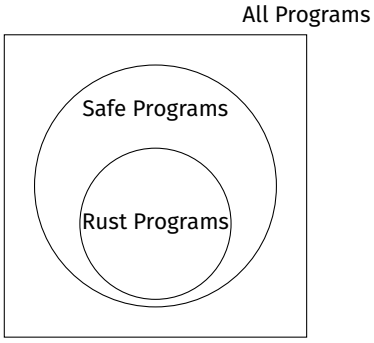
These are examples of unsafe programs and they all have at least one thing in common: a program having unwanted behavior. Thus safety could be thought of as a part of correctness of a program. We made proofs about correctness of a program when we did Operational Semantics. However, OpSem is not the only way to describe meaning of a program, and it cannot capture or measure all the semantics or behavior of a program.

In the above examples and what Rust hopes to mitigate, is the insecurity of memory. Human insecurity (like falling for a phishing attack, or being tortured, not insecurity like self esteem (however this can also be leveraged)), is not something Rust or any language can really prevent. Memory safety has a few types of incorrect behavior, for example:

- Giving read access when it shouldn't (sending plaintext passwords)
- Gives write access when it shouldn't (overwriting people's grades)

- Denies read access when it shouldn't (DOS)

There are others but these are 3 big ones. If Rust can prevent unwanted behavior, then it has succeed in it's job. However, making sure a program is secure is very hard. Thus Rust will take a very conservative approach: It will refuse to compile programs it is unsure if they are safe or not. This is sometimes known as whitelisting: denying everything except what you know. This is opposed to blacklisting: allowing everything except what you don't want. Whitelisting is more secure. What this means: there are a set of programs that are safe, but Rust's compiler cannot verify it so Rust will not compile your safe program. Visually:



14.2.2 Stack and Safety

As we saw in the Garbage Collection Notes, memory safety comes down the idea that memory is not being managed correctly. We also said the garbage collector makes languages slow. So we need some fast way of determining when and how to manage memory. To do so, let's look at another place in memory: the stack.

When we need to allocate memory on the stack, we simply push the value onto the stack. When we need to deallocate memory, we simply pop it off. In order to allocate something on the stack we need to know 2 major things. The first is more obvious: we need to know how much memory we need to allocate. This is why things that have a fixed size link an `int` will be stored on the stack, but things of changing size are stored on the heap. The second is less obvious: we need to know how long that item needs to live in memory. Consider function calls. The parameters and local variables of the function have a scope. We know when the item enters memory (when the function is called) and we know when it needs to leave (when the function returns). This is why when we have items that need to exist for longer than a certain scope need to return a pointer, rather than the value.

Both of these points allow for the automatic memory management of the stack: we know how much to `malloc` since size is known, and we know exactly when to `free` due to knowing when the scope ends. Now items on the stack are not `malloc`'d or `free`'d but the point still stands, to automatically mangage memeory, we need to know these 2 things. Rust will introduce concepts built into the design of the language to allow for this behavior.

14.3 Statements vs Expressions and Codeblocks

Before we can begin to talk about how rust manages memory without a garbage collector, we need to know some syntax for code examples later used.

To begin, in Rust, there is a explicit distinction between expressions and statements which we need to consider. As we saw in OCaml, an expression is something that evaluates down to a value. A statement on the other hand does not evaluate to a value but may perform some sort of action.

For example, something like `let x = 5` is a statement, not an expression. It does not evaluate to a value. You could not say something like `3 + (let x = 5)`. However things like 3 or 1+2 are expressions, and they evaluate to the value of 3. Expressions can be part of other expressions: `3 + (2 + 3)`, and they can be part of expressions: `let x = 3 = 4`.

This becomes important because of the idea of a codeblock which Rust allows. Rust allows for this idea of a codeblock which is a collection of statements followed by zero or more expressions surrounded by curly braces. For example:

```
1 let x = 3;
2 {
3   let y = 5;
4   let z = 7;
5   println!("{}",y+z);
6 };
```

Lines 2-6 are the codeblock. This codeblock has three statements and zero expressions. Statements are ended with a semicolon while expressions are not.

```
1 let x = 3;
2 {
3   1 + 2;
4   3 + 5
5 }
```

Lines 2-5 are the codeblock. This codeblock has one statement: `1 + 2`, and one expression: `3 = 5`. The codeblock itself is treated as an expression.

Again, codeblocks are zero or more statements followed by at most one expression. If we were to describe the structure with some regex syntax, we could say that a codeblock looks like

`{ stmt;* expr?  }`

Some more example of codeblocks:

```
1 {
2     1 + 2;
3     let a = 4 - 3;
4     a
5 }
```

two statements, one expression

```
1 {
2     1 + 2;
3     let a = 4 - 3;
4     a;
5 }
```

three statements, zero expressions

```
1 {
2     1 - 3
3 }
```

Zero Statements, one expression

Codeblocks are expressions themselves which means we can capture the value of the last statement in a codeblock:

```
1 let a = {
2   let b = 3;
3   let c = 4;
4   b + c
5 };
6 println!("{}",a);
```

Here we can see a codeblock on lines 2-4 which has two statements and one expression, where the result of the expression is being bound to the a variable. This however raises the question: what happens when there is no expression?

```
1 let a = {let b = 3; b + 5;}
2 println!("{}",a);
```

In this case, Rust will use the default return value, which is called `unit`. Like Ocaml, `unit` is an empty tuple: `()`. Thus the above code will fail (since Rust does not not know how to print the `unit` type). We could instead however do the following:

```
1 let a = {let b = 3; b + 5;}
2 if a == (){
3     println!("a is unit type");
4 }
```

This is a perfect segue to using codeblocks as expressions in the `if` expression.

14.4 If Expression

The `if` expression is exactly that: an expression. Thus it evaluates to a value. Much like Ocaml, each branch of the `if` expression must return the same type. You may be thinking, but what about that very last example in the above section? We will get to that.

The `if` expression takes the form of `if guard_expr {true_block} else {false_block}`. The `true_block` and `false_block` are both codeblocks whereas the `guard_expr` is an expression (which could also be a codeblock). For example:

```
if true {false} else {true}
// analogous to Ocaml's expression: if true then false esle true

if false {
    let a = 3;
    let b = 4;
    println!("{} is not {}",a,b)
} else {
    println!("I am false")
}
// First no such thing as multiline comment
// Second: this shows the true codeblock having two statements and 1 expression
//         whereas the false codeblock has 1 expression
//         (technically println! is a macro that exapnds to an expression)

if {let a = 1;
    let b = 3;
    a < b}
    {0}
    else
    {1}
// example where the guard is a codeblock (because a codeblock is an expression)
// the true and false block are just one expresssion
// analagous to Ocaml's: if
//                         (let a = 1 in
//                         let b = 3 in
//                         a < b)
//                         then 0 else 1
```

In each of these examples, the type of true block is the same as the type of the false block. Breaking this type check will fail to compile:

```
if true {3} else {"hello"}

if true {3}
```

This second example is a case of the `if` expression without an else block that we saw as the last example in the previous section. That example does, work, but this one does not. Why?

We know that a codeblock with no expression will evaluate to the `unit` type by default. The same is the case here. When the `else` block is left out, then the default return type of the else branch is type `unit`. Thus, if we want to leave out the `else` block and still be able to compile, we need to make sure the `true_block` will also evaluate to type `unit`. This can be done by in a few ways: we can make the last expression a statement by adding a semicolon, or we use an expression that returns type `unit`.

```
if true {3;}
// here the true block has one statement and zero expressions
```

```
// and hence evaluates to unit

if true {3; println!("println! is an function that evaluates to unit")}
// println! is an function that evaluates to unit

if true {3; ()}
// We could explictly return unit
```

14.5 A bit of data types and Functions

Functions in Rust are expressions, they evaluate (return) a value (which includes unit). Functions are a named collection of commands which are dependent on an input (an empty input is included here). They can also be unnamed (anonymous functions) which Rust call closures. To define a function however we first need to talk about data types.

14.5.1 Data Types

Built-in data in Rust has type which can be thought of as scalar (flat) or compound. These are pretentious words that describe types based on what is needed to describe the data. Scalar types include things like numbers, booleans, and characters. Since Rust wants to have some of the advantages of C which include fine grain control of memory, Rust breaks it's numeric types of integers and floats into subtypes. That is to say, there is no `integer` type, but rather "integer of 8 bits", "integer of 16 bits", "integer of 32 bits", etc. See below the table of data types:

Integers		
Size	Signed	Unsigned
8 bits	i8	u8
16 bits	i16	u16
32 bits	i32	u32
64 bits	i64	u64
128 bits	i128	u128
Machine Dependent	isize	usize

Machine dependent sizes are analogous to the `sizeof(int)` since different machines have different architectures (32-bit vs 64-bit for example). For Floats, there are just `f32` and `f64`.

Last of the scalar types are `bool` and `char`. It is important to note, that characters are 4 bytes long so they include more than ascii, including utf-8 characters (which includes emoji and characters in other languages).

Compound types on the other hand, are pieces of data that need at least 2 values to define its type. In Rust, the built in compound types are tuples and arrays. Like OCaml, tuples in Rust are fixed size, hetergenous and defined by the types it holds. A `(u8,u16)` tuple is different from a `(u16,u8)` tuple and both are different from a `(u8,u16,u32)` tuple. Again, the empty tuple is called `unit` and used to say it the value is nothing meaningful.

Arrays on the other hand are different from arrays in other languages or lists in OCaml. They must be both homogenous and a fixed length. An array's type is defined by the type of data it holds and its length.

```
let a = [1,2,3,4]
// this has type [i32;4]. Rust will default to i32 for numbers in that range
let a = [3;5]
// this tells Rust to make an arrayof size 5 with each element being the value 3
// this has type [i32;5]
```

When describing a type of a value or variable, the colon notation is used.

```
3: i32
```

```
'h': char
true: bool
[1,2,3]: [i32;3]
(1,1.0,false): (i32, f64, bool) //default float is f64
```

14.5.2 Functions

Now that we know some data type notation, we can now talk about functions! Functions can have zero or more parameters, and will return a single value (could be `unit`). In Rust we need to annotate types of our inputs and output if they are not `unit`.

```
fn main(){
    println!("Hello")
}
// this function takes in input and returns unit. Notice I don't need a semicolon here

fn this(x: i32){
    println!("{}", x + 4);
}
// this function takes in a single argument of type i32 and returns unit. Do not need to
    denote the return type but do need to specify the input type

fn that(x:u8, y:u8) -> bool{
    x > y
}
// this function takes in two arguments and returns a boolean. Need to denote the return and
     input types since they are not unit

fn the_other_thing() -> char{
    'a'
}
// this function takes in zero arguments and returns a char. Need to denote the return since
     it is not unit
```

14.5.3 Closures

We can make anonymous functions called closures. Their notation looks like a Ruby Codeblock.

```
|x| x + 1
// parameters are surrounded by |
// body of closure is an expression

|x, y| x + y
|x, y| {let z = x + y; z}
// expression can be a codeblock
|x:i32| -> i32 {x}
```

There are a few important things to note

- Much like Python and OCaml, we can bind a closure to a variable (`let x = |y| y`).
- Rust will perform type inference on a closure so type annotations are not needed (but they are prefered!)
- when using a return type annotation, you need to put the expression in a codeblock (for the parser)
- Closures cannot use generics. So `|x| x` cannot be called on both a int and string. It will default to the type of the first call

14.6 Ownership

We are now finally ready to start talking about the thing that makes Rust safe: ownership. Ownership describes a set of rules to determine when a value is going to be `drop`'d (Rust's equivalent to `free`[2]). It can also influence who has read/write access. The rules of ownership is as follows:

- Every value in Rust has an Owner
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped

The last rule here is very important: it tells you when a value should be `free`'d which was one of the rules to help bring automatic memory management to Rust without a garbage collector. What exactly is a an owner though? An owner is the one responsible for freeing and is the one who can access the value. This can be better seen here:

```
1 let x = 3;
2 {
3   let a = 4;
4 }
5 println!("{}",x);
```

Here, 3 is a value and x is the owner. The owner goes out of scope after line 5 so it is dropped then. 4 is also a value, and it's owner is a. a goes out of scope at line 4 so it is dropped then. However, this is ultimately a tad misleading and doesn't really showcase anything since they are stored on the stack to begin with. They don't really have complicated interactions with ownership. Let us talk about values stored on the heap. For example: a `String`.

The `String` data type is part of the standard library, not something built-in to Rust and can change size, hence it must be stored on the heap. String literals however are hardcoded in memory and are technically owned by the Rust program. Regardless, let's take a look at this heap allocated `String` and how ownership affects it.

```
1 let s = String::from("Hello");
2 //this is calling the 'from' function from the 'String' library. Namespacing
```

Here there is the value "hello" stored on the heap and its owner is 's'. This is uninteresting so let's see something fun:

```
1 let s = String::from("Hello");
2 let x = s;
3 println!("{}",s); //fails to compile
```

This fails to compile. This is because of the second rule of ownership. Only 1 owner is allowed at a time. When we execute line 2, ownership of "Hello" is **moved** to the variable 'x'. Think of it this way: I own a jar of dirt. If I give it to you, then I cannot use the jar of dirt again since you now have ownership of it. Thus, if we wanted to print "Hello", we would have to do the following:

```
1 let s = String::from("Hello");
2 let x = s;
3 println!("{}",x);
```

Why does Rust make you do this? Consider the following:

```
1 char* s = malloc(sizeof(char) * 6);
2 char* x = s;
```

When we no longer need that char pointer, who should free that segment of memory? 'x' or 's'? We don't want them both to free, that would be a double free. Rust gets around this by making sure it knows who exactly can free and knows when. This also prevents the use after free since 's' becomes invalid once ownership is moved to 'x' (so s cannot use that memory address), and the value is dropped once 'x' goes out of scope (hence it becomes impossible for x to use that memory address due to scoping constraints).

[2] Drop is actually a function that is called when 'freeing' so it's more like a deconstructor in C++

This process of passing ownership is called moving. The owner of a value can be moved around using let bindings, function calls, closure creation, etc. With closures you need to explicitly tell rust to move ownership and I would say you probably won't come across this in this course. For let bindings and function calls, let's consider the following:

```
1 let x = String::from("Hello"); //x owns value
2 let y = x; // y now owns, x becomes invalid
3 let z = y; // z now owns, y becomes invalid
4 let a = String::from("Hello"); // a now owns it's own copy of hello
5 let b = a; // b now owns the second hello, a is invalid
6 println!("{} is {}", z, b); // this is fine
```

In this scenario, 'x' initially points to a value of "Hello". After passing ownership to 'y' which then passes ownership to 'z', a new segment of memory is allocated with the value of "Hello" and 'a' becomes the owner of this second instance of "Hello". 'z' still maintains ownership of the first instance of "Hello" since it's a separate segment of memory. Thus, this code compiles because the rules are followed, each value has its own owner.

We can see this more in action when we add codeblocks, which change the scope of variables.

```
1 let x = String::from("Hello"); //x owns value
2 {
3   let y = x;
4   println!("y is the owner of {}",y);
5 };
6 println!("x cannot be used here");
```

in this case, 'x' is the inital owner of "Hello", but then 'y' takes ownership in line 3. When the owner 'y' goes out of scope at line 5, the value is dropped. Ownership is **NOT** passed back to 'x'. Thus, we cannot use 'x' in line 6.

For functions, we pass ownership into functions via parameters and pass ownership back using return values. Consider:

```
1 fn main(){
2   let x = String::from("Hello");
3   let y = this(x);
4   println!("y is the owner of {}",y);
5 }
6
7 fn this(a:String) -> String{
8   a
9 }
```

In this case, if we consider a code trace of the above, we could say that the lines of execution would be something like

- Line 1: main is called
- Line 2: 'x' becomes owner of the newly made "Hello" String
- Line 3a: the `this` function is called in line 3
- Line 7: function is called and stack frame is made
- Line 8: 'a' is evaluated and returned
- Line 9: stack frame is popped off
- Line 3b: return value from `this` is bound to 'y'
- Line 4: print line is run
- Line 5: main is returned, program ends

This is a rough estimation of the order in which lines are executed. We can trace this order to figure out how ownership of "Hello" is passed around.

- Line 2: 'x' is the owner of "Hello"
- Line 3a/7: ownership of "Hello" is moved from 'x' to 'a' during this function call. 'x' becomes invalid
- Line 8,9,3b: ownership of "Hello" is moved from 'a' to the variable capturing the return value; 'y'. 'a' becomes invalid but is dropped anyway because the function ends.
- Line 4: Since 'y' is the owner, this is valid
- Line 5: the owner 'y' goes out of scope so "Hello" is freed now

This is important to note because the following would not work because "Hello" gets dropped when the function ends.

```
1 fn main(){
2   let x = String::from("Hello");
3   that(x);
4   println!("{}",x); // will not compile
5 }
6
7 fn that(a:String) -> String{
8   a
9 }
```

Here since 'a' is the owner and goes out of scope with nothing capturing the return value, "Hello" is dropped around line 9, after line 3 executes.

14.6.1 No Heap Values, Copy trait

This whole process of ownership and passing ownership around really only affects values on the heap. Values on the stack are just immediately copied. See the following:

```
1 let x = 3;
2 let y = x;
3 println!("{} == {}", x,y);
4
5 let a = String::from("Hello");
6 let b = a;
```

In the above example, line 3 is valid since many built in data types support the Copy Trait. A `Trait` is analagous to an interface in Java. More on these later. For now, just know that instead of 'y' getting ownership of the value that 'x' owns, a copy of 'x' is made and given to 'y'. Thus, 'x' and 'y' are both owners of their own instance of '3'. This is analogous to a previous example where we had two variables that pointed to their own segment in memory even if the value was the same.

Any struct (object) in Rust that has the Copy `trait` (implements the copy interface) will instead copy the value (using a memcpy) so there will be two instances of the value, each with their own owner. Copy is not overloadable and happens implicitly, if you wanted finer control of how your data is copied (like when making your own data structure), use the `clone` trait). More on this later

14.7 Borrowing

In most cases it makes no sense to pass ownership to a function, especially if you want ownership back. Consider the following:

```
fn main(){
  let x = String::from("hello");
  let(y,z) = get_len(x);
  println!("{} has len {}",z,y);
}

fn get_len(a:String)-> (usize,String){
  let l = a.len();
  (l,a)
}
```

In this case, if I wanted to get the length of a value, I don't need to give full ownership of the String, because then I have to then get ownership back. It should be sufficient to *borrow* the data. That is, if I have a jar of dirt, you can take a look at it, but it's still mine. I could even lend it to you to look at and but something in, but I want it back. Rust allows this occur with the idea of references and borrowing.

However if we are lending things out, we need to make sure we maintain the ability to know when to drop things, and know what can use pieces of data so there isn't insecurities. To help us out, there are two(ish) rules of references:

- Rule 1: Every reference must be valid (we need to say this. We will see later)
- One but not both (XOR) of the following must will be true:
  - Rule 2a: You can have any number of immutable references
  - Rule 2b: You can have one mutable reference

Before we begin, we need to talk about mutability. In Rust, every variable is immutable. That means once you bind a value to a variable, you cannot change that value. A let binding is a new binding every single time, so like OCaml, the variable is shadowing any previously made variable of the same name. To make a variable mutable, you can use the `mut` keyword.

```
let x = 3;
let x = 4; // new binding, shadows the previous line

let mut y = 5;
y = 6;
```

The above is an example of the shadowing and `mut` keyword use. Just because you use the `mut` keyword, does not mean you have to use it mutably. This will become important when talking about references.

So back to references. A reference is like a pointer that has certain access rights. Ownership is like a special type of reference that will invalidate any previous references if possible. Otherwise we can make references that don't take ownership and have certain access rights to the value in question. Let's see what this means:

```
let x = String::from("Hello");
{
    let y = &x; // & tells rust to have y borrow from x, not take ownership
    println!("I can use {} and {}",x,y)
};
println!("I can still use {}",x);
```

In this case, in Line 1, 'x' is the owner of "Hello". In line 3, 'y' borrows from 'x', making rule 2a be true: there are some number (two in this case, x and y) of immutable references to "Hello". When we have immutable references, we have read access to that piece of data so we can use both 'x' and 'y' to read "Hello". When 'y' goes out of scope on line 5, the value of "Hello" is not dropped since 'y' was not the owner, allowing us to still use 'x' on line 6. Using this idea, we can better make use of our `get_len` function:

```
fn main(){
  let x = String::from("hello");
```

```
3   let y = get_len(&x);
4   println!("{} has len {}",x,y);
5 }
6
7 fn get_len(a:&String)-> usize{
8   a.len()
9 }
```

In this case, the variable 'a' in `get_len` does not take ownership of "Hello" but instead receives a read-only reference to it. Thus, we can still use 'x' on line 4 even when the function that uses the reference ends and 'a' goes out of scope.

So back to the rules. Rule 1 is pretty straightforward, we cannot have dangling pointers. Rule 1 prevents this from occuring. Consider the dangling reference in the C code:

```
1 int* x = this();
2
3 int* this(){
4     int i = 5;
5     return &i;
6 }
```

in this case, 'x' is dangling since the place it points to was dropped when the `this` function ended. This is prevented in Rust by ensuring that all references are valid. Rust will not compile the following:

```
1 fn main(){
2   let s = that();
3 }
4 fn that()->&String{
5     let s = String::from("Hello");
6     &s
7 }
```

This also doesn't really make sense in Rust, just give ownership, no need to pass a reference.

In regards to rule 2: it is important to note, that we can have any number of immutable references at a time (as long as there are no mutable references):

```
1 let x = String::from("Hello");
2 let y = &x;
3 let z = &x;
```

it is also important to note, that Rust will try its best to dereference automatically through deref coercion. This is important when talking about smart pointers (we will get to this), but for now it means we can do the following:

```
1 let x = String::from("Hello");
2 let y = &x;
3 let z = &x;
4 let a = &z;
5 let b = z;
6 println!("I can use all these values to read {},{},{},{},{}",x,y,z,a,b);
```

Here, rust is automatically dereferencing lines 4 and 5 so they point to the "Hello" value in memory. This has to deal with the `Deref` trait for those interested.

Back on track, what about mutable references? When we have a mutable variable, we can make either immutable references or mutable references to it. We cannot make a mutable references to an initially immutable variable. We also have to make sure we keep rule 2 in mind as we do this. Consider:

```
1 let mut x = String::from("Hello");
2 x.push_str(" World"); // concats " World" to "Hello"
3 {
```

```
4       let y = &x;
5       println!("{} and {} can only read, no write",x,y);
6   };
7   x.push_str("!");
8   println!("{} is still valid",x);
```

At line 1: 'x' is the mutable owner of the "Hello" value so it can read and write to "Hello". We can see it write in line 2 with the `push_str` function. Then at line 4, an immutable reference is created, which makes us take a look at rule 2. We cannot have both one immutable and one mutable reference to a value so what happens? In this case, Rust makes the 'x' mutable reference immutable (revokes write access) for the entire duration of 'y's lifetime (not scope!). This means, we cannot mutate the now "Hello World" string until 'y's lifetime ends (at line 5). Afterwards, 'x' gains write access again and is allowed to mutate "Hello World" to "Hello World!" in line 7.

The purpose behind this rule is to prevent dangling pointers, and also prevent data races. Data races are a good place for undesired behaviour to occur, and temporal attacks, while difficult to pull off, can be devastating. To mitagate data races, the rules of references come into play. Let us rephrase rule 2 into read and write access terms:

- You can have many readers to a piece of data and no writers XOR
- You can have 1 writer and no readers to a piece of data at a time.

This means nothing could read a piece of data as it is being updated, and no more than one thing could write at a time. Thus, many data races are mitigated in Rust which make for secure programs. There are ways to do concurrency correctly with `Mutex` which we will get to (//TODO).

14.8 Lifetimes

As we just saw, I made a distinction about lifetimes and scope. Rust does the same thing. Scope in Rust is no different than any other language, it determines where a variable could be used. Rust however goes a step further and makes a distinction about a variable's lifetime: when is the variable actually used? In many cases, the lifetime and scope of a variable are the same, but there are also many cases when they are different. Consider:

```
1  { let x = 3;
2    { let y = 4;
3      println!("{}",y);
4    }
5    println!("Hello World");
6    let z = 5;
7  }
```

In this example, 'x's scope is lines 1-7, 'y's scope is lines 2-4, and 'z's scope is line 6-7. However, 'x' is only used on line 1, making it's lifetime Line 1. 'y's lifetime is the same as it's scope: it is used first on line 2 and last used on line 3/4, and 'z's lifetime is also the same as its scope: line 6/7.

Let's see this in terms of mutable and immutable references:

```
1  let mut x = String::from("Hello");
2  let y = &x;
3  println!("{}",y);
4  x.push_str(" World");
```

In this case: 'x' starts off mutable, but during 'y's lifetime, becomes immutable, so once 'y's lifetime ends, 'x' becomes mutable again. 'y's lifetime is lines 2 nd 3 so after line 3, 'x' becomes mutable again. If we swap lines 3 and 4, this will not compile since 'x' only becomes mutable after 'y's lifetime ends.

```
1  let mut x = String::from("Hello");
2  let y = &x;
3  x.push_str(" World");
4  println!("{}",y); // this does not compile
```

Lifetimes are actually part of a reference's type in Rust. Thus, when we use references in functions, we need to make sure we know their lifetimes. A function with the type signature `fn this(x:&'a i32)` would not accept a piece of data that has a lifetime of `'b`. Now we can't explicitly define a lifetime in Rust, but we can compare lifetimes to each other. We can say that two variables 'a' and 'b' have either different lifetimes, or the same lifetime. We use a generic modifier like we did in OCaml: 'a, 'b, etc. For the most part we no longer have to manually annotate lifetimes on references since Rust has some handy rules for determining lifetimes through type inference. This is done with an extension to the type checker called the Borrow checker. Rust will run this checker before compilation and try to figure out lifetimes. If Rust cannot determine the lifetimes through the borrow checker, you need to help the compiler out by annotating lifetimes for references. If the borrow checker sees a contradiction of lifetimes, then Rust will not compile your program. Let's first look at the rules of lifetimes and then we can see examples:

- Rule 1: For every parameter that is a reference: we assign a new lifetime generic
- Rule 2: if there is exactly one input reference, if the output is a reference, we assign it to the same lifetime as the parameter
- Rule 3: If there are more than one input reference but one of them is `&self` or `&mut self`, then if the output is a reference, it recieves the same lifetime as the `self` parameter.

Let's consider the following function headers:

```
fn this(x: &i32, y: &i32, z: &i32) -> i32
// Rust will use rule 1 here to give each parameter a different lifetime
// -> fn this<'a,'b,'c>(x: &'a i32, y: &'b i32, z: &'c i32) -> i32
// <> after a function means it will use a generic. Much like Arraylist<T> uses a generic
// More on generics later

fn that(x: &i32,y:i32) -> &i32
// Rust will use rule 1 to give every input reference a different life time.
// -> fn that<'a>(x: &'a i32,y:i32) -> &i32
// Notice that 'y' doesn't get a lifetime. That is because it's not a reference
// Rust will then use rule 2 to give the output reference a lifetime
// -> fn that<'a>(x: &'a i32,y:i32) -> &'a i32

fn otherthing(&self, x: &i32, y:i32) -> &i32)
// Here &self is a reference to an object. Like in puython where you have self to refer to
// the current object, Rust also uses self. This means we have things like Objects and
// structs in Rust.
// Using rule 1 we give every input reference a lifetime
// -> fn otherthing<'a,'b>(&'a self, x: &'b i32, y:i32) -> &i32)
// Rule 2 does not apply here since there are more than one input refernce
// Rules 3 does apply however so we can assign the output reference a lifetime
// -> fn otherthing<'a,'b>(&'a self, x: &'b i32, y:i32) -> &'a i32)
```

However, there are cases when these rules cannot cover or determine the lifetimes of all references. What happens if we have multiple input references, but none are `self`? In these cases, we need to manually give lifetimes to lifetimes to the parameters.

```
fn this(x: &i32, y: &i32) -> &i32
// Rust will attempt to use rule 1 and give a lifetime to every input
// -> fn this<'a, 'b>(x: &'a i32, y: &'b i32) -> &i32
// Rule 2 does not apply, nor does rule 3.
// We cannot determine the output's lifetime so Rust Fails and makes use
// put explicit lifetimes on the parameters.
// either of the following would work
fn this<'a>(x: &'a i32, y: &'a i32) -> &'a i32
fn this<'a,'b>(x: &'a i32, y: &'b i32) -> &'a i32
```

```
// The following would not work since there would be no way for the
// function to make a new lifetime without breaking rule 1 of references
fn this<'a,'b,'c>(x: &'a i32, y: &'b i32) -> &'c i32
```

14.8.1 Dangling Pointers

So now that we have an idea about what a lifetime is, and how Rust infers a lifetime for a reference, we should probably see why Rust decided to make rules for this.

It all stems from the first rule of references: that all references must be valid. In order to determine if a reference is valid, Rust needed some way to determine if a reference was actually pointing to live data, or data that is invalid. Consider the following C Code:

```
int main(){
  char* s1 = malloc(sizeof(char)*6);
  strcpy(s1,"hello");
  char* l;
  {
    char* s2 = malloc(sizeof(char)*4);
    strcpy(s2,"bye");
    l= longest(s1,s2);
    free(s2);
  }
  printf("%s is longer",l);
}

char* longest(char* x, char* y){
  if (strlen(x) > strlen(y)){
    return x;
  }else{
    return y;
  }
}
```

In this program, all pointers will be valid and you will not have any memory safety issues. I could however, change s2 to make this program unsafe:

```
int main(){
  char* s1 = malloc(sizeof(char)*6);
  strcpy(s1,"hello");
  char* l;
  {
    char* s2 = malloc(sizeof(char)*10);
    strcpy(s2,"byebyebye");
    l= longest(s1,s2);
    free(s2);
  }
  printf("%s is longer",l);
}

char* longest(char* x, char* y){
  if (strlen(x) > strlen(y)){
    return x;
  }else{
    return y;
```

```
  }
}
```

Here, `l` is a pointer trying to point to the value of `'byebyebye"` but that memory area has already been freed. If we are lucky, we will get a segfault.

This small change can make a program safe or unsafe. Rust will fix this problem and enforce the former using lifetimes. Let's convert this to Rust without explicit lifetimes real quick:

```
fn main(){
  let s1 = String::from("hello");
  let long;
  {
    let s2 = String::from("bye");
    long = longest(&s1,&s2);
  }
  println!("{} is longer",long);
}

fn longest(x:&str, y:&str) -> &str{
  if x.len() > y.len() {x} else {y}
}
```

During compilation, Rust's borrow checker will make sure that all references are valid and try to infer the lifetime (type) of all the references. Let's take a look at `longest`:

```
fn longest(x:&str, y:&str) -> &str
// Rule 1: give each input reference a different lifetime:
-> fn longest<'a,'b>(x:&'a str, y:&'b str) -> &str
// Rule 2 does not apply: there is more than one input reference
// Rule 3 does not apply because there is no self.
```

If we try to apply the above rules, Rust will be confused as to what the return type of the `longest` function is so it will not compile this program.

So we need to manually add the annotations of the lifetimes. We could try a few ways:

```
fn longest<'a,'b,'c>(x:&'a str, y:&'b str) -> &'c str
// If we try to do this, we will not compile because a lifetime of 'c could only be created
// in the function leading to a dangling pointer

fn longest<'a,'b>(x:&'a str, y:&'b str) -> &'a str
// this is valid, but recall that a lifetime is part of type. This means we could not
// return y, because y (&'b str) has a different type than the the return value (&'a str)
fn longest<'a>(x:&'a str, y:&'a str) -> &'a str
// this is valid and would allow us to return either x or y
```

As noted in the comments, only real way to add lifetimes for a generic longest string function would be the last way (`fn longest<'a>(x:&'a str, y:&'a str) -> &'a str`). However, how does that impact our program?

```
fn main(){
  let s1 = String::from("hello");
  let long;
  {
    let s2 = String::from("bye");
    long = longest(&s1,&s2);
  }
  println!("{} is longer",long);
```

```
}

fn longest<'a>(x:&'a str, y:&'a str) -> &'a str{
  if x.len() > y.len() {x} else {y}
}
```

This means that both inputs to `longest` have to live at least as long as the return value. This is not true: while `s1` lives at least as long as the return variable `long`, `s2` does not live this long. So Rust will not compile this program (even though we know this particular program is safe). This is an example of why Rust has a steep learning curve and why you will be fighting the compiler on some fronts. It also showcases that Rust uses a conservative approach to safety.

14.8.2 Structs and Traits

14.8.3 Smart Pointers

Appendix A

NFA to DFA

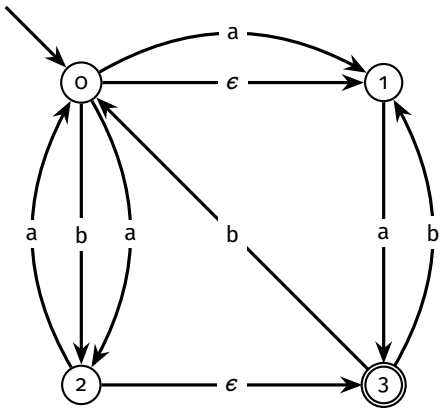
We will walk through the NFA to DFA algorithm step by step. Let's first consider the subset contruction algorithm:

```
NFA = (a, states, start,finals,transitions)
DFA = (a, states, start,finals,transitions)
visited = []
let DFA.start = e-closure(start), add to DFA.states
while visited != DFA.states
  add an unvisited state, s, to visited
  for each char in a
    E = move(s)
    e = e-closure(E)
    if e not in DFA.states
      add e to DFA.states
    add (s,char,e) to DFA.transitions
DFA.final = {r| r∈ DFA.states and ∃ s ∈ r and s ∈ NFA.final}
```

For this example: let's take the following NFA:

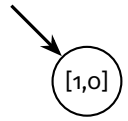
```
NFA = (['a','b'], // the alphabet
       [0,1,2,3], // the states
       0,         // the starting state
       [3],       // the final states
       [(0,"",1), // the transitions in the form of (source,char,destination)
       (0,'a',1), (0,'a',2), (0,'b',2),
       (1,'a',3), (2,'a',0), (2,"",3),
       (3,'b',0), (3,'b',1)]
```

This NFA would visually look like the following:



To begin, the DFA has not been constructed but we should at least know the alphabet for the DFA. So we can at this moment in time that `DFA = (['a','b'], [],?,[],[])`. Next we set `visited = []`. We have now done the first 3 steps of the algorithm and are ready to begin building our DFA.

The first step is to figure out what the DFA start state is. In this case, `DFA.start = e-closure(start)`. So in this case, we take the NFA's start state (state 0) and perform `e-closure` on it. In this case, we ask ourselves, where can I go from state 0 using any number of ϵ transitions and only ϵ transitions? Here we can only go to state 1, and since every state has an implicit ϵ transition to itself, we can say that `e-closure(1) == [0,1]`. We now add that to our DFA states. Our result looks something like:




```
DFA = (['a','b'],   // the alphabet
       [[1,0]],     // the states
       [1,0],       // the starting state
       [],          // the final states
       []           // transitions
```

We know how to take a look at the next step of the algorithm. The next step is: