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

[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.

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:

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

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

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

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

easier to write with because they have shortcuts or macros that cover the hard and tedious stuff. Recall the assembly project in 216:

```
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

- How to express and learn the syntax for a programming language
- Learn how to use some features that languages have and how they work
- Knowing how language features work, being able to implement them in any language

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.

```ruby
# hello_world.rb
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

```
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.

```
# program1-2.rb
variable1 = 5
variable2 = 4
variable3 = variable1 + variable2
print variable1
print " + "
print variable2
print " = " # gross. We will learn to convert later
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

```
# program1-3.rb
variable1 = 5
variable2 = 4
variable3 = variable1 + variable2
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:

```ruby
# checking.rb
def add(a,b)
    puts a + b
end

add(1,2)
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`

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

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.

```
# oop.rb
puts 3.class
puts "Hello".class
puts 4.5.class
```

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.

```
# oop.rb
puts 3.class
puts 3.class.ancestors

a = "hello"
b = a
puts a.equal?(b)        #true
puts a.equal?("hello")  #false
puts "hello" == "hello" #true
```

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

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.

```
# nil.rb
puts nil.methods
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:

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

    def count
        @@count
    end
    # ... same as before ...
end
s = Square.new(5)
s2 = Square.new(6)
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.

```
# counter.rb
class Counter
    @@count = 0
    def initialize()
        @@count += 1
    end

    def Counter.counter
        @@count
    end
end
c = Counter.new
c1 = Counter.new
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:

```
# return.rb
def to_s1
    s = "Hello"
    return s
```

```ruby
end

def to_s2
    s = "Hello"
    s
end

def to_s3
    "Hello"
end
 puts to_s1
 puts to_s2
 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.

```ruby
 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

```ruby
# overload-err.rb
class Square
    def func1(x)
        puts "func1"
    end

    def func1(x,y)
        puts "func2"
    end
end
Square.new.func1(2,3) #fine
Square.new.func1(2)   #error
```

But we could do something like the following

```
# override.rb
class Square
    def func1(x)
        puts "func1"
    end

    def func1(x)
        puts "func2"
    end
end
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.

```
# double.rb
puts 3.methods.include?(:double)

class Integer
  def double
    self + self
  end
end

puts 3.methods.include?(:double)
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

```
# break.rb
class Integer
  def +(x)
    "Not Today"
  end

  def -(x)
    self * x
  end
end

puts 3+4
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:

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

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

```
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.

```ruby
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.

```ruby
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

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

But because we cannot instantiate we cannot do

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

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

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

```ruby
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:

```ruby
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.

```ruby
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.

```
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.

```
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?

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

```
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.

```
# 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`.

```
# 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.

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

```ruby

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 = []
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
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:

```
 1  # conditional.rb
 2  count = 1
 3  while count >= 0
 4      if 3 > 4 then # then is optional
 5          puts hello
 6      elsif nil
 7          puts "nil is true"
 8      else
 9          if count == 0
10              puts FalseClass == false
11          end
12          puts NilClass == false
13      end
14      count -= 1
15  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

Regular Expressions

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

Chatbot+9000

Despite Regular expressions being something that is language independent, you will need to know how to use them in a language, and since this is taught in the Ruby section of the course, we will be learning how to use Ruby's regular expressions. There will then be a smaller section on how to use them in OCaml, but that's more syntactical than anything really substantive. 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.

3.1 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,

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

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.

3.2 Regular Expression Basics

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. 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.

- Grouping - Ultimately not something that is the basis of regex, but is helpful in giving precedence to the above.

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

3.3 Regular Expression In Ruby

Now that we have an idea of what regular expressions are, let us see how we can use them in ruby, and what type of patterns we can create. Ruby and many other language support the POSIX-EXT standard of regular expressions which is what we will be going over here. Let's start out on just 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: `/`.

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

Here is the very first example of a valid regular expression pattern. 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 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. Suppose we have our earlier pattern.

```ruby
# 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. If the pattern is found, it will return the index of the first character of the first instance of the pattern.

Regex Syntax

Ultimately this pattern is not fun nor interesting so let's take a look at other patterns and what they match. POSIX-EXT standard (and Ruby) regular expressions have the following pattern description/syntax.

- Ranges: To talk about accepting a range of characters you can use the following: `/[a-z]/`. This particular example means accept a lowercase letter from a through z. You can of course modify this for uppercase: `/[A-Z]/`, digits: `/[0-9]/`, or smaller ranges `/[A-F]/`. Ultimately any ascii range works: `[!-&]`. It is important to note that this will only match a single character.

- Concatenation: we already saw this but if `/[a-z]/` matches a single character then `/[a-z][a-z]/` will match two consecutive lowercase letters. Technically `/pattern/` matches the seven character literals 'p', 'a', 't', 't', 'e', 'r', 'n' in a row.

- Union: To match string that may have alternative spellings, you may use the union character: `/ste(v|ph)en/`. This may be useful for alternative spellings. Be careful because the scope of the union operator goes until a parenthesis or the beginning or end of the regex. That is, `/abc|def/` will match either "abc" or "def" whereas `/ab(c|d)ef/` will match either "abcef" or "abdef". If you have many things being unioned together, you can use the bracket syntax. `[/[aeiou'/]` will match the same strings as `/a|e|i|o|u/`. Technically `/[a-z]/` is shorthand for `/a|b|c|...|y|z/`

or /[abc..xz]/. This also means you can combine ranges. /[a-zA-Z0-9]/ will match any alphanumeric character.

- Repetitions: A pattern is typically thought as something repeating so of course there is a repeating operation. There are three types of repetitions that are supported:
  - 0 or more times: Represented as a $*$, this is placed to say that the previous pattern can occur 0 or more times. For example: /0*[0-9]/ could match any single digit number with any number of preceding zeros. (eg. "0001", "2", "00", "0", "0000000009" would all be matched).
  - 1 or more times: Represented as a +, this is placed to say that the previous pattern can occur 1 or more times. For example: /[0-9]+/ would match any number ≥ 0. (eg. "0", "1", "10", "123","54234543" would all be matched).
  - Exact or bounded repeats: Represented as {x},{x,y},{x,},{,y} you can make sure than a certain of number of repeats occurs. For example /[0-9]{2}/ matched only 2 digit numbers like "00,"20","99". 0-9]{3,} will match any number of at least 3 digits, whereas /[0-9]{,3}/ will only match numbers that have at most 3 digits. Lastly /[0-9]{2,3}/ will match numbers of only 2 or 3 digits. These bounds are inclusive.
  - 0 or 1 repeats: Represented as ?, this will check if the pattern is there or not. For example /-?[0-9]+/ will match positive or negative integers.

  It is important to note that all of these modifiers only apply to the immediately preceding pattern. That is to say /cliff*/ and /(cliff)*/ will match different things. You can use parenthesises if you wish to extend the scope of any of these operators.

- Negation: If you want to say *anything but* a specific character, you can do so in the brackets with the carrot symbol. For example /[^b][a-z]+/ says any word at least 2 characters long that does not start with the letter 'b'.

- Wild card: Represented as ., this will match any character. That is, despite how it looks, the pattern /[0-9].[0-9]{2}/ would match on both "3.30" and "3:30".

- Capture Groups: see next page.

If you wish to use any of these special characters as a literal, you can escape them in the regex. That is, to actually match a float with 2 decimal places you can do so with the pattern /[0-9]+\.[0-9]{2}/.

Additionally, using the = method will search the entire string for a match. If you wish, you can make sure that it starts matching at the beginning of the string with the ˆ operator or the end of the string with the $ operator. Consider the following:

```
# 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`.

Capture Groups

Now that you can check to see if a string matches a pattern, we can move onto the more useful part: extracting out parts of a string. Suppose that you are given phone number such as (301) 405-1000[2]. If you just need the area code, I suppose you could do something like `phone_number[1..3]`. But what if you have a text file of phone numbers that look like

```
(111)-111-1111
222-2222222
3333333333
```

Then we would need to use a pattern to match what each line could be, and find a way of extracting the area code. Regex makes this easy by having you surround the pattern you wish to capture with parenthesis. Consider the following:

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

By placing the area code in parenthesis, regex knows to store the string that matches the pattern `/[0-9]3/`. Now where does regex store it? For Ruby, they are stored in top level variables which can be accessed with the $. So $1 referes to the first capture group. If you had multiple capture groups like `/([0-9]3)-([0-9]7)/` then $2 would refer to the next captre group and so on. I am unsure how many capture groups can exist with this method.

One thing to be careful about is the fact that parenthesis are also used for scoping of things like the ∗ operator so those are also captured. Capture groups are ordered by when the opening parenthesis occurs so you can nest capture groups like so: `/(([0-9])[0-9]2)-[0-9]7/`.

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

This will print out 123 and then 1. 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 despute how easy it is.

[2]This is the university's phone number

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.

```ruby
# 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])*$/
```

3.4 Regular Expression In OCaml

Chapter 4

OCaml

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

Cleff

This is another 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 Ruby 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.

4.1 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 supports both declarative and imperative programming. We will talk about this all in the next section, but for now let's just write our very first program.

```
(* helloWorld.ml *)
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.

4.2 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 want to not 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.

4.2.1 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
```

```
    Declarative
+ Take all the values that a
+ 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 ruby we get something like

```ruby
#imperative.rb
results = []
arr.each{|item|
    remainder = item % 2
    if remainder == 0
        results.push(0)
    end
}
results
```

Now we haven't learned enough (or really any) OCaml at this point to write a solution to this yet[1] , but let's look at a declariative python example:

```python
#declarative.py
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.

4.2.2 Side effects and Immutability

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

```ruby
# side_effects.rb
@count = 0
def f(node)
  node.data = @count
  @count+=1
  @count
end
```

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

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

`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**. Referentail transparency is the ability to 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?"

4.2.3 Expressions and Values

In functional languages, one of the core ideas is ability to treat functions as data. Which means much like Ruby Procs, we can pass them in as arguments, or use them as return values in methods. 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. 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 (* is a value, has type bool *)
3 * 4 (* is an expression, has type int *)
"hello" ^ "world" (* is an expression of type string *)
5.4 (* 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. 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` is not an expression and just binds an expression to a variable. 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))
```

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 $p \vee q$.

4.2.4 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 Ruby or C, the only things that evaluate to `bool`s 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
```

4.2.5 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
puts x

def x
    3
end
puts 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 *)
```

Similar to `let bindings` a function definition by itself is not an expression, but the variable that is bound to the function is. 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. On the other hand, something like `let area l w = l * w` is not an expression but a binding of a function to variable.

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`?

4.2.6 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 `floats` 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
```

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  (* ('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`

4.3 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`.

4.3.1 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.

4.3.2 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).

4.3.3 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.

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

4.3.4 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:

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

```
let rec sum lst = match lst with
    [] -> 0
    |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 4.4).

4.4 Data Types and Syntax

4.4.1 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 `Rect`s which hold `int * int` tuple information, or `Circle`s which hold `float`s. 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
*)
```

4.4.2 Syntax

Chapter 5

Higher Order Functions

The headquarters of the Order of the Phoenix may be found at number twelve, Grimmauld Place, London.

Harry Potter

5.1 Intro

We cover this topic in OCaml so the examples here will be mostly written in OCaml.

5.2 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 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;
}

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

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

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.

5.3 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 saw 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 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 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;;
```

5.4 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. The syntax of an anonymous function is

```
(* add1 *)
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) *)
```

This is no different that just saying something like `2 + 3` instead of saying something like (* let x = 2 + 3 *). 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 *)
```

Which means `let add1 x = fun 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`?

5.5 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:

```
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 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.

5.6 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`.

5.7 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.

5.7.1 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 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]
*)
```

5.7.2 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 5.8) about why this we can minimize the number of stack frame, but since you already know how fold works, here is `fold_left`

```
let rec fold f a l = match l with
[] -> a
|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.

5.8 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

```c
int fact1(int x){
  if (x == 1)
    return 1;
  return -1
}
int fact2(int x){
  if (x == 2)
    return 2 * fact1(x-1);
  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

```
int fact4(int x){}
    if (x == 1)
        return 1;
    if (x <= 4)
        return x * fact4(x-1);
    return -1;
}
```

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 6

Finite State Machines

Finite State Machines? More like infinite town devices

Cringe

6.1 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 quick but perhaps 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`.

```
#define FALSE 0
#define TRUE 1
```

For a more complicated example if I wanted to add pattern matching in C, then maybe I create something like the following:

```
// something to hold pattern matched data
struct Data{
  ...
}
// need to create a new match function
// takes in the data to match
// takes in list of (data -> int (read bool)) functions
// take in a list of functions that take in data to use and return
    anything (void pointer)
void∗ match(data∗ value, int (∗∗patterns)(data∗), void∗ (∗∗exprs)(
    data∗))
```

This represents a new matchable data type, and a new match function.[1]. This way is nice for simple things, but for complex ways, it's a bit of a hassle to work with. At that point, you might as well use a different language. Which is exactly the point: you should either use a different language, or you can always add something to the language. To add a feature to a language, you need to do so 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!).

6.1.1 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 to create what we call tokens. 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. To do so, we will need to find the link between our machine and a language.

6.1.2 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. 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 *ands*, *ors* and *negates* 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) but typically FA is used in the context of abstract theory and FSM is used with the context of an actual machine, but they all 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

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

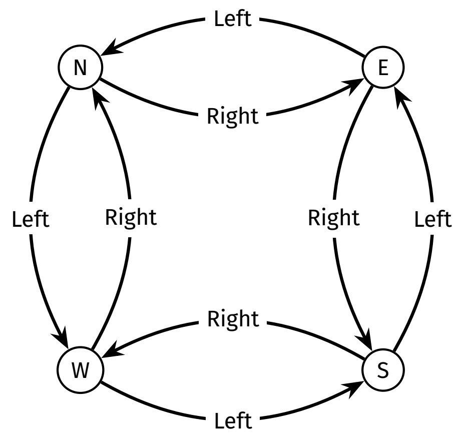
[2]Initially called an 'a-machine' or atomic machine by Alan Turing.

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 need something more.

6.1.3 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, an 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 face 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. This is not a good example to show what a starting or accepting state is, but we will see that in the next section.

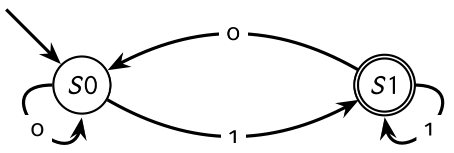
The important takeaway is **Based on where I am (which state), and what action occurs (which edge I choose), I can tell you where I will end up**. So given an input, and a series of instructions, I can give you an output (sound familiar?). For example, If I start at state N, and my instructions are to 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).

6.2 Regex

So we did this whole thing with graphs and talked about what a machine is and a single example a 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 a letter of the alphabet. That is 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 had an understood starting state (N), but not really 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 when starting at the starting state (denoted by an arrow with no origin), after 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 second character 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 expression) does accept the string `"00101"`.

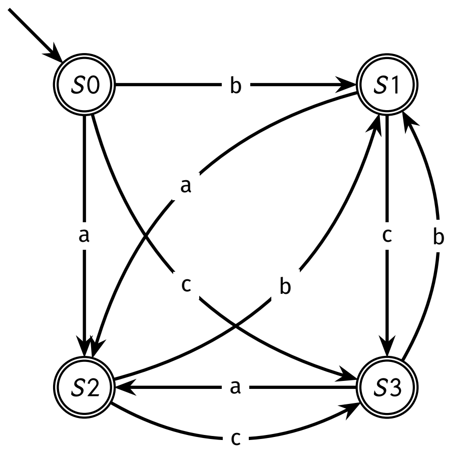
One final thing: a FSM for regex only will tell you if a string is accepted or not[3]. It will not do capture groups, will not tell you if the input is invalid, it will only tell you if the string is accepted (in the case of invalid input, it will tell you the string is not accepted).

6.3 Deterministic Finite Automata

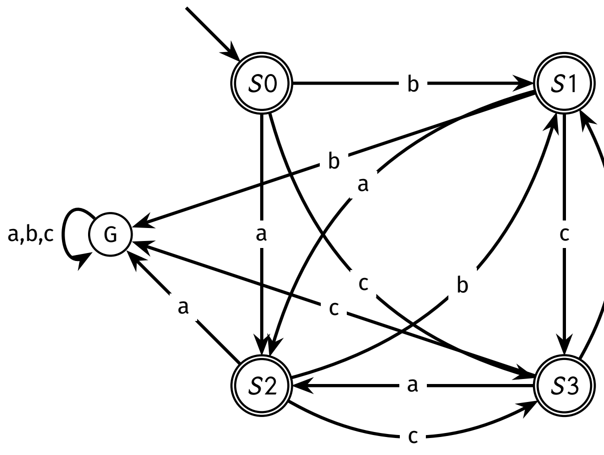
All FSMs can be described as either deterministic or non-deterministic. So far we have seen only deterministic finite automata (DFA). If something is deterministic (typically called a deterministic system), then that means there is no randomness or uncertainty about what is happening (the state of the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

[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



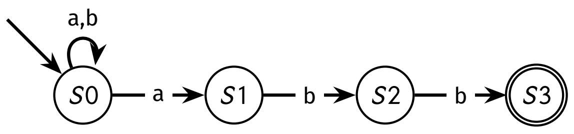
Now this graph is missing a few states (one really). What happens when I am in state $S0$ and I see a "b"? There is an implicit state which we call a "garbage" or "trash" state. A trash state is a non-accepting state where once you enter, you do not leave. There is an implicit one if you are trying to 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.

6.4 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:

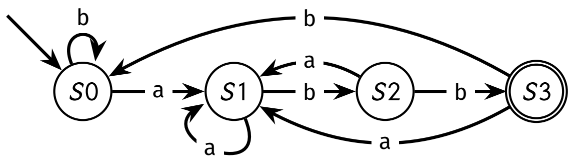


This machine represents the regular expression /(a|b)*abb/. There is still a starting state, transitions, ending states, all the things we see for a FSM. However, there is something interesting when we look at the transitions out of $S0$. If I am looking at the string "abb", then when I am traversing, 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
```

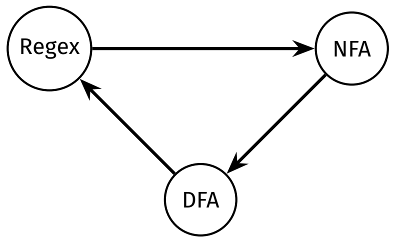
Since the traversal of the graph is uncertain, we call this non-deterministic. To check acceptance for an NFA, we have to try every single valid path and if at least one of them ends in an accepting state, then we accept the string. Since we have to check all possible paths, you can imagine that this 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:



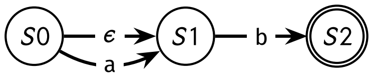
It is much easier to go from a regular expression to an NFA than it is to go from a regular expression to a DFA. Additionally, NFAs, because they can be more condensed, are typically more spatially efficient than their DFA counterpart. However, there is of course a downside: NFA to regex is difficult, 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 "B" may look like:

S0 -a-> S1 -b-> S2

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

6.5 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:

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

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^*)

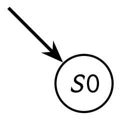
To convert from a regular expression to a NFA, all we need to figure out how to represent each of these things as an NFA. Since this grammar is recursive, we will start with the base cases, and then move on to the recursive definitions.

6.5.1 Base Cases

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

The $\emptyset$

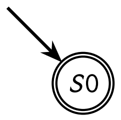
The empty set is a regex that accepts nothing (the set of strings (the language) it accepts is empty). This machine can be constructed as just the following:



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)

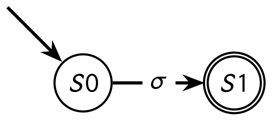
The regular expression that only accepts the empty string means the set of accepted strings is {""}. This set is not empty so it is different from $\emptyset$. This machine can be constructed as the following:



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.

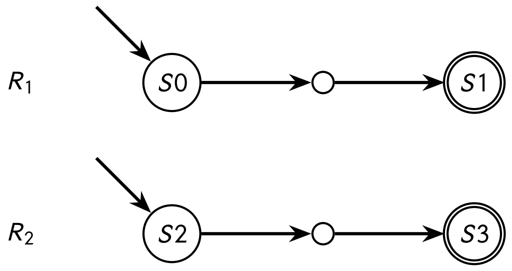
A single character σ

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:



6.5.2 Concatenation ($R_1 R_2$)

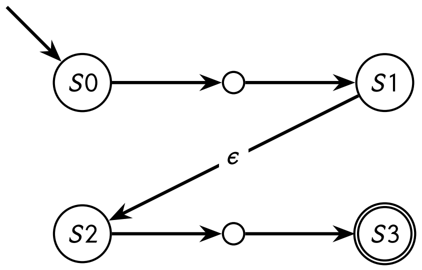
Now that we have our base cases, we can begin to start showing how to do a recursive operation. Aside from the $\emptyset$, each base case has a starting state and an accepting state (sometimes these are the same state). Now since the $\emptyset$ is empty, all the recursive definitions cannot rely on 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.



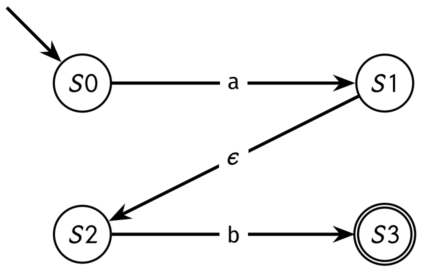
Now we don't know what regular expression R_1 and R_2 are, just that they have 1 starting state, and 1 accepting state. The small, unlabeled nodes here just represent any internal nodes could exist (if any).

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:



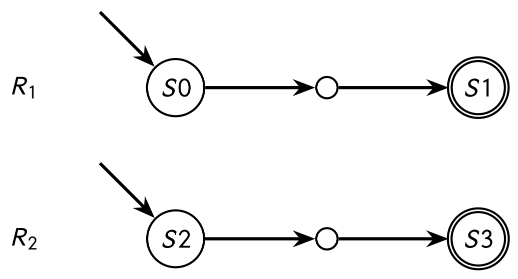
To take our previous example of R_1 being /a/ and R_2 being /b/, when we want to concatenate these machines we get the following machine:



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.

6.5.3 Branching ($R_1|R_2$)

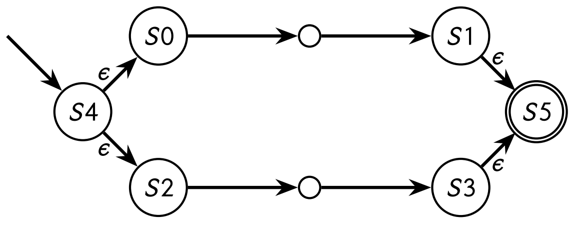
Branching or union is the next recursive definition and requires a bit more work than our concatenation. Again, let us assume that we have some previous regular expressions R_1 and R_2 that have a starting state and an accepting state.



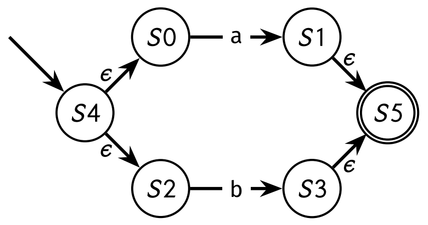
Again we don't know what regular expressions R_1 and R_2 are, just that they have one starting state and one accepting state.

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:



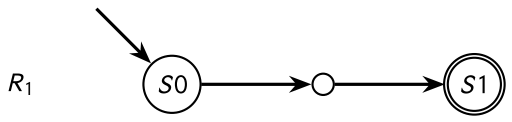
This new machine still has one starting state, and one accepting state which means we can inductively build larger machines, and the ϵ-transitions allow us to chose either path or go to the accepting state without consuming anything. To take our previous example of R_1 being /a/ and R_2 being /b/, when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



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

6.5.4 Kleene Closure (R_1^*)

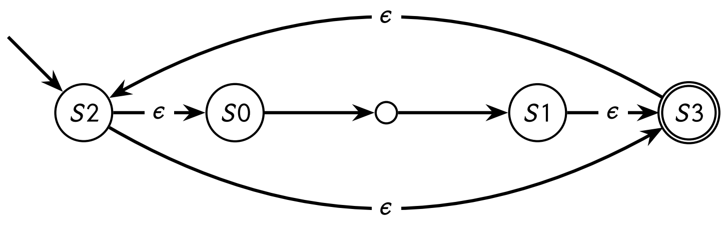
Kleene closure gives us the ability to repeat patterns infinitely many times and is the last recursive definition of a regular expression. Despite having the ability to infinitely repeat, it will look very similar to our union machine. Additionally, this is the only recursive definition that does not 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.



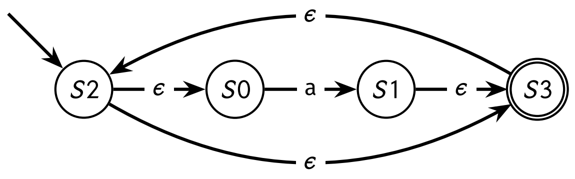
Again we don't know what regular expression R_1 is, just that it has one starting state and one accepting state.

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 $\{x|x = \epsilon \vee x \in L_1 \vee x \in L_1L_1 \vee x \in L_1L_1L_1 \vee \ldots\}$. For example, if R_1 is /a/ then $L_1 = \{"a"\}$ and we are looking for /a∗/ or $L_2 = \{"", "a", "aa", "aaa", \ldots\}$.

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
```

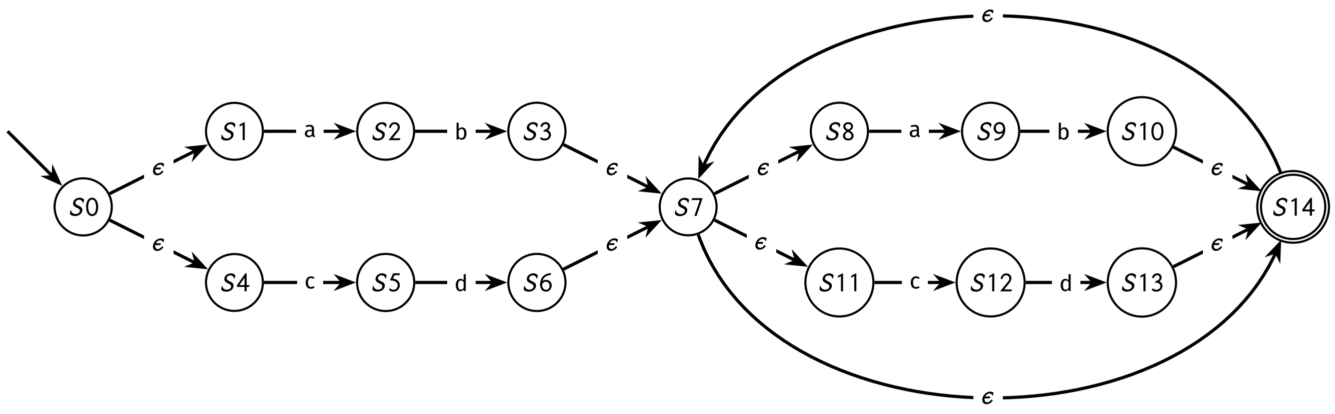
If I wanted to accept `"aa"`, then my traversal would look like

```
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.

6.5.5 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:

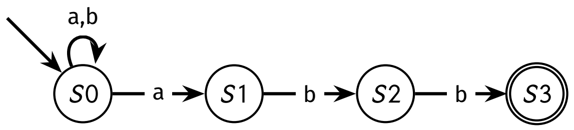


We could optimize this even further, but not really needed. Additionally to see without any optimizations and for a step-by-step, you can see Appendix B (//TODO).

6.6 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:

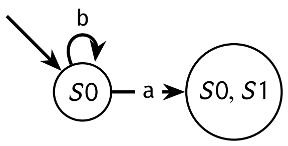


When we want to check acceptance, we need find all possible paths and check if at least one accepts it. That is when checking if the machine accepts `"aabb"`, we need to check all of the following paths:

```
      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 $S1$).

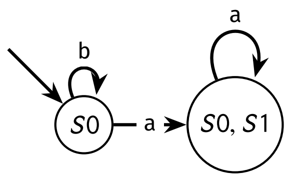


Now while we have a new state that shows possible states I could be in after seeing a `"a"`, I then need to figure out what to do next. That is, if I am in this new state $S0$, $S1$, then what happens if I see a `"a"` or a `"b"`?

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 $S0$

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

or $S1$. Additionally, (looking at the original 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:



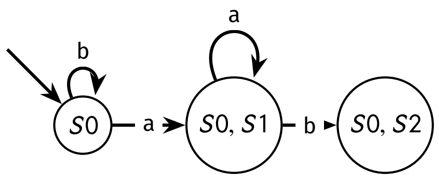
If this is confusing, consider the following logical argument:

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

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).

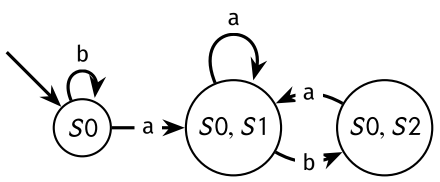
$$\begin{array}{rl} & S0 \Rightarrow \{S0, S1\} \\ & S1 \Rightarrow \varnothing \\ & S0 \vee S1 \\ \hline \therefore & \{S0, S1\} \cup \varnothing = \{S0, S1\} \end{array}$$

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 $S2$.

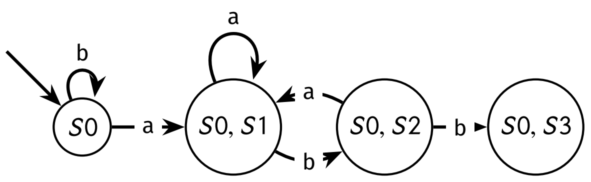


But now the issue continues, if I am in state $S0$, $S2$, and see a symbol, I do not know where I should go. SO let's continue with considering if I was in $S0$ or $S2$ and seeing either a "a" or a "b".

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.

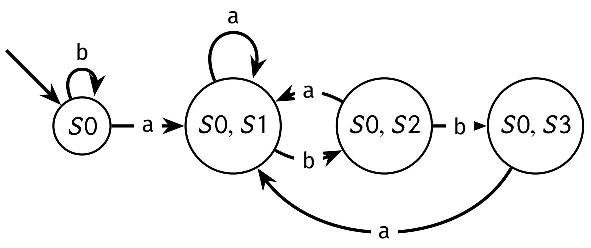


Now if I am in state $S0$ and see a "b" then I can only go to state $S0$. If I am in state $S2$ and see a "b", then I can only go to state $S3$. So if I am in either state $S0$ or $S2$, then I will end up in either $S0$ or $S3$. Here is another place of uncertainly so let us add this new state with transition.

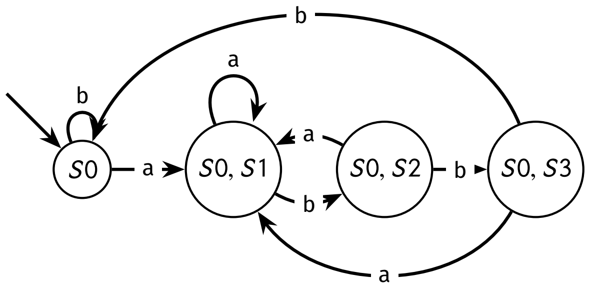


Again, the same issue arises. If I am in state $S0, S3$, what happens if I see a "a" or "b"? Well we will have to calculate this like we did with the other states.

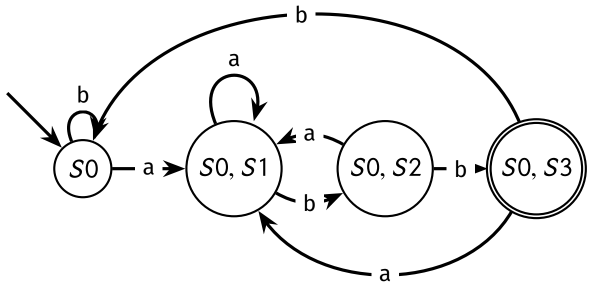
If I am in state $S0$ and see a "a", then I could either be in $S0$ or $S1$. If I am in state $S3$ and see a "a", then I can go nowhere (except a trash state). So if I am in either state $S0$ or $S1$, then I can only end up in $S0$ or $S1$. Let us add this transition.



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.

6.6.1 NFA To DFA Algorithm

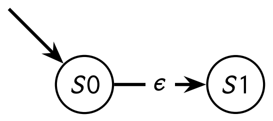
So let us convert what we just did to an algorithm.

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 definition 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

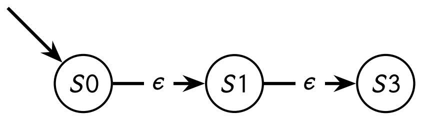
Now to our functions. Recall that we want to figure out which states can be grouped together. ϵ-closure is a function that helps figure this out in terms of ϵ-transitions. The previous example did not have any ϵ-transitions but consider:



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.

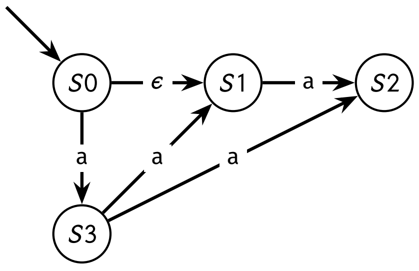
The actual pseudocode is something like:

```
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| s \in r and s \in NFA.final}
```

For a full in-depth example, see Appendix C //TODO

6.7 DFA to Regex

Chapter 7

Operational Semantics

I am not a fully operational person

Cliff

7.1 Introduction

Now that we know a few languages, let's dive a little bit deeper in how languages functions.

- 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.

7.2 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?

[1]Would recommend Phil360: Philosophy of Language with Alexander Williams

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 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.

7.3 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 $((p \land q) \land (p \Rightarrow r) \land (q \Rightarrow r)) \rightarrow r$. Namely, we can make a proof:

$$\begin{array}{rl} & p \land q \\ & p \Rightarrow r \\ & q \Rightarrow r \\ \hline \therefore & r \end{array}$$

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.

7.4 Operational Semantics

Let us define a very basic language $Alanguge$:

$$\begin{array}{rl} e \rightarrow & n \\ \rightarrow & e\,?\,e \\ n \rightarrow & 0|1|2|3|\ldots \end{array}$$

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.

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{e_1 \mathbin{?} e_2 \Rightarrow n_3}$$

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 $e_1 \mathbin{?} e_2$ 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:

$$\frac{\frac{}{4 \Rightarrow 4} \qquad \frac{}{3 \Rightarrow 3} \qquad 7 \text{ is } 4 + 3}{4 \mathbin{?} 3 \Rightarrow 7}$$

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:

$$\frac{\frac{}{3 \Rightarrow 3} \qquad \frac{\frac{}{4 \Rightarrow 4} \qquad \frac{}{5 \Rightarrow 5} \qquad 9 \text{ is } 4+5}{4 \mathbin{?} 5 \Rightarrow 9} \qquad 12 \text{ is } 3 + 9}{3 \mathbin{?} 4 \mathbin{?} 5 \Rightarrow 12}$$

As we add more to our language, we need to add more rules to our operational semantics. Let us consider the language $Blanguage$:

$$\begin{array}{rcl} e & \longrightarrow & n \\ & \longrightarrow & e + e \\ & \longrightarrow & V \\ & \longrightarrow & \text{let } V = e \text{ in } e \\ n & \longrightarrow & 0|1|2|3|\ldots \\ V & \Rightarrow & a|b|c|d|\ldots \end{array}$$

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 $e \longrightarrow V$? 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 $e \longrightarrow \text{let } V = e_1 \text{ in } e_2$.

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

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad A, x : 3; x + 4 \Rightarrow e_3}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We then want to use our plus rule when evaluating $x + 4$

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{A,x{:}3;e_4 \rightarrow n_1 \qquad A,x{:}3;e_5 \rightarrow n_2 \qquad n_3 \text{ is } n_1+n_2}{A,x{:}3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Here we can do what we did above an notice that in $x + 4$ that x is e_4 and 4 is e_5.

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{A,x{:}3;x \rightarrow n_1 \qquad A,x{:}3;4 \rightarrow n_2 \qquad n_3 \text{ is } n_1+n_2}{A,x{:}3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Based on our variable lookup rule we can say that $x \Rightarrow 3$ making $n_1 = 3$:

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x{:}3;(x) \Rightarrow 3}{A,x{:}3;x \rightarrow 3} \qquad A,x{:}3;4 \rightarrow n_2 \qquad n_3 \text{ is } 3+n_2}{A,x{:}3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We know that 4 evaluates to itself:

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x{:}3;(x) \Rightarrow 3}{A,x{:}3;x \rightarrow 3} \qquad \frac{}{A,x{:}3;4 \rightarrow 4} \qquad n_3 \text{ is } 3+4}{A,x{:}3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

and we know that $3 + 4$ is the value of 7:

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x{:}3;(x) \Rightarrow 3}{A,x{:}3;x \rightarrow 3} \qquad \frac{}{A,x{:}3;4 \rightarrow 4} \qquad 7 \text{ is } 3+4}{A,x{:}3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Thus we know that e_3 is the final value of the expression.

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x{:}3;(x) \Rightarrow 3}{A,x{:}3;x \rightarrow 3} \qquad \frac{}{A,x{:}3;4 \rightarrow 4} \qquad 7 \text{ is } 3+4}{A,x{:}3;x+4 \Rightarrow 7}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow 7}$$

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):

$$\begin{aligned} e \longrightarrow &\quad n \\ \longrightarrow &\quad e + e \\ \longrightarrow &\quad V \\ \longrightarrow &\quad \text{let } V = e \text{ in } e \\ \longrightarrow &\quad B \\ \longrightarrow &\quad \text{if } e \text{ then } e \text{ else } e \\ n \longrightarrow &\quad 0|1|2|3|\dots \\ V \Longrightarrow &\quad a|b|c|d|\dots B \Longrightarrow true|false \end{aligned}$$

We should add some rules to incorporate these new values.

TODO, but enough to post notes

7.5 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 $Alangauge$

$$\frac{}{A; n \Rightarrow n}$$

$$\frac{A; e_1 \Rightarrow n_1 \qquad A; e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A; e_1 + e_2 \Rightarrow n_3}$$

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:

```
def rec eval expr env= match expr with
 Num(x) -> x
|Add(e1,e2) -> let n1 = eval e1 env in
               let n2 = eval e2 env in
               let n3 = n1 + n2 in
               n3;;
```

In moving to $BLanguage$, we gain more rules:

$$\frac{A(x) \Rightarrow v}{A; x \Rightarrow v}$$

$$\frac{A; e_1 \Rightarrow v \qquad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow e_3}$$

We can then update our interpreter accordingly:

```
def rec eval expr env = match expr with
 Num(x) -> x
|Add(e1,e2) ->   let n1 = eval e1 env in
                 let n2 = eval e2 env in
                 let n3 = n1 + n2 in
                 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;;
```

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.