CMSC330 Notes

Cliff
Contents

1 Intro
   1.1 What is a language? ................................................. 3
   1.2 How do we use language? ........................................ 3
   1.3 Why so many languages? Why use one over another? ........ 4
   1.4 Language Features .............................................. 4
   1.5 Conclusion .................................................... 5

2 Ruby
   2.1 Introduction .................................................. 7
   2.2 Typing ......................................................... 8
   2.3 Object Oriented Programming ................................. 10
      2.3.1 Class Creation .......................................... 11
   2.4 Code blocks .................................................. 16
   2.5 Modules ..................................................... 18
   2.6 Data Types and Syntax ......................................... 20
      2.6.1 Numbers ................................................ 20
      2.6.2 Strings and Symbols ................................... 21
      2.6.3 Arrays .................................................. 21
      2.6.4 Control Flow .......................................... 23

3 Higher Order Functions ........................................... 25
   3.1 Intro ......................................................... 25
   3.2 Functions as we know them .................................... 25
   3.3 Higher Order Programming ................................. 26
   3.4 Anonymous Functions ....................................... 27
   3.5 Partial Applications ........................................ 27
   3.6 Closures .................................................... 28
   3.7 Map .......................................................... 29
      3.7.1 Map .................................................... 29

4 Regular Expressions .............................................. 33
   4.1 Introduction ................................................ 33
   4.2 Regular Expression Basics ................................... 33
   4.3 Regular Expression In Ruby .................................. 34
   4.4 Regular Expression In OCaml ............................... 37

5 Regular Expressions ............................................... 39
   5.1 Introduction ................................................ 39
   5.2 Regular Expression as a Tool ............................... 39
   5.3 Regular Expression Creation ............................... 40
   5.4 Regular Expression In Ruby .................................. 40
   5.5 Regular Expression In OCaml ............................... 43
### 6 OCaml

6.1 Introduction .................................................. 45
6.2 Functional Programming ........................................... 46
   6.2.1 Declarative Languages ........................................ 46
   6.2.2 Side effects and Immutability ............................... 47
   6.2.3 Expressions and Values ...................................... 47
   6.2.4 The if Expressions .......................................... 48
   6.2.5 Functions as Expressions ................................... 49
   6.2.6 Type Inference ................................................ 49
6.3 OCaml Pattern Matching .......................................... 50
   6.3.1 Lists ............................................................. 50
   6.3.2 Recursion ...................................................... 51
   6.3.3 Pattern Matching ............................................ 51
   6.3.4 Recursive Functions ......................................... 52
6.4 Data Types and Syntax ........................................... 53
   6.4.1 Data Types .................................................... 53
   6.4.2 Syntax .......................................................... 55

### 7 Higher Order Functions

7.1 Intro ............................................................... 57
7.2 Functions as we know them ...................................... 57
7.3 Higher Programming .............................................. 58
7.4 Anonymous Functions ............................................ 59
7.5 Partial Applications ............................................. 59
7.6 Closures ........................................................... 60
7.7 Common HOFs ..................................................... 61
   7.7.1 Map ............................................................. 61
   7.7.2 Fold ............................................................ 62
7.8 Tail Call Optimization ........................................... 64

### 8 Finite State Machines

8.1 Introduction ..................................................... 69
   8.1.1 Compilers ..................................................... 69
   8.1.2 Background - Automata Theory ............................. 69
   8.1.3 Finite State Machines ....................................... 70
8.2 Regex .............................................................. 71
8.3 Deterministic Finite Automata .................................. 72
8.4 Nondeterministic Finite Automata .............................. 72
8.5 Regex to NFA ..................................................... 74
   8.5.1 Base Cases ..................................................... 74
   8.5.2 Concatenation \((R_1R_2)\) .................................... 75
   8.5.3 Branching \((R_1|R_2)\) ....................................... 76
   8.5.4 Kleene Closure \((R_1^*)\) ................................. 76
   8.5.5 Example ....................................................... 77
8.6 NFA to DFA ........................................................ 78
   8.6.1 NFA To DFA Algorithm ..................................... 81
8.7 DFA to Regex ..................................................... 82

### 9 Languages

9.1 Introduction ..................................................... 83
9.2 Context-Free Grammars .......................................... 83
9.3 Designing Grammars ............................................. 85
   9.3.1 Regular Expressions Supported ............................ 85
   9.3.2 Not supported by Regular Expressions .................... 86
   9.3.3 A basic Grammar ........................................... 86
## Contents

9.4 Modeling Grammars .................................................. 87

10 Interpreters and Compilers ......................................... 91
  10.1 Introduction ......................................................... 91
  10.2 Compilers/Interpreters .......................................... 91
  10.3 Lexing ............................................................... 92
  10.4 Parsing .............................................................. 93
  10.5 Evaluating/Generating ........................................... 94

11 Operational Semantics ............................................. 97
  11.1 Introduction ......................................................... 97
  11.2 Meaning ........................................................... 98
  11.3 Correctness ......................................................... 98
  11.4 Operational Semantics .......................................... 98
  11.5 Definitions interpreter ........................................... 101

12 Lambda Calculus ..................................................... 103
  12.1 Intro ............................................................... 103
Chapter 1

Intro

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 HESP 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, 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. To be put even more succinctly: a language is a medium used to express ourselves. And in my experience, programming languages are no different. However it's not that simple.

1.2 How do we use language?

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

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

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

---

1 I would recommend taking philosophy of language with Alexander Williams
2 Thomas Nagel has a fun little paper called "What is it like to be a bat" that says no matter how much we know about bats and no matter how hard we imagine what echolocation would be like, we could not experience how a bat navigates.
3 Some would say this sounds like grammar. This is partially true since grammar is a subset of syntax
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!\n");
```

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

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

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

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

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

```
func();
```

That is most languages have some way to implement or represent a function call (typically means adding parentheses 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:
1.5 Conclusion

This course focuses on a few things

- Recognizing language features and analyzing their effects on problem solving
- Imperative and functional programming
- Computational abilities of regular, context-free and Turing-complete languages
- Deriving meaning from a language
- Creating proofs about the correctness of a program
- Designing languages and implementing their evaluation
Chapter 2

Ruby

A regretful honey maker could be called a rue-bee

Kliff

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

## 2.1 Introduction

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

```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

```bash
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.

```ruby
# program1.rb
variable1 = 5
variable1 = "hello"
variable2 = 4
variable3 = variable1 + variable2
puts variable3
```

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

```
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.

```ruby
# program1-1.rb
variable1 = 5
variable2 = 4
variable3 = variable1 + variable2
puts variable3
```

Now if we run our code we get a different error. We get `undefined local variable or method `variable3' 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.

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

```ruby
# 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 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:

```c
#include <stdio.h>

int main()
{
    int x;
    x = "hello"
    return 0;
}
```

Should we try to compile this code with the `-Werror` 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` 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. 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.

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

1. Canonically it will just raise an error and still compile
2. We will see how ruby does this when we talk about parsing. In particular Project 4
You can see that everything, including primitives, are object oriented. Also notice that like java, when calling an object’s methods, we use the dot syntax. That means that earlier when we called a + b in add.rb, it was actually doing something like a.+(b). Don’t believe me? Consider the following:

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

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

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

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

```ruby
# nil.rb
puts nil.methods
puts nil.to_s
```

### 2.3.1 Class Creation

Let’s make our first class

```ruby
# square.rb
class Square
  def initialize(size)
    @size = size
  end

  def area
    @size*@size
  end
end

s = Square.new(5)
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.
// 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

# square-1.rb
class Square
    # ...
    def size
        @size
    end
    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:

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

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

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

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

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

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

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

```ruby
# 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.

```ruby
# 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

```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:

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

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

```ruby
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:

```ruby
def func
  if block_given?
    yield
  end
end

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:

```ruby
def func1
  yield 5
end

def func2(i)
  y = i+1
  puts y
end

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

Right off the bat, we should acknowledge that codeblocks can take in parameters when yielded to, as seen on line 2. The syntax for accepting arguments can be shown on line 10, where you surround 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 func2 can take in a variable parameter, but the code executed will always be the same. Let's see this more clearly:

```ruby
# 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.

```ruby
# 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.

```ruby
# 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.

```ruby
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 each other. For example
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:
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:

module Doubler
    def Doubler.base
        2
    end

    def double
        self + self
    end
end

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

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

That's it. Pretty simple. We cannot extend modules but we can still overwrite them; But we are getting ahead of ourselves. Let's just first see how we use them.
class Integer
  include Doubler
end
puts 10.double
Here we are adding the Doubler module to the Integer class so any integer now has access to the doubler method. We can also do things like
puts Doubler.base
puts Doubler.class
puts Doubler.instance_methods
But because we cannot instantiate we cannot do
Doubler.new
Doubler.double

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

module M1
  def bye
    "Goodbye"
  end
end
module M2
  def bye
    "Bye"
  end
end
class C
  include M1
  include M2
end
puts C.new.bye
In this case, we load modules in the order we include them so M/two.osf has the last instance of defining bye so M2's bye method will be called. Had we swapped the order:
class C
  include M2
  include M1
end
puts C.new.bye
Then M1's bye method would be called instead. If we had an instance method in the C class, then we would call C's bye method.
class C
  include M2
  include M1
  def bye
    "C ya"
  end
end
Typically the order in which something is called is by first looking at self, then the self's modules, then the parent's instance methods, then the parent's modules, then the grandparent's instance methods, then the grandparent's modules, etc. We can see that here:
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
  include M1
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 Strings 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
```

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"]
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 to 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.
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 let’s 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:

# conditional.rb

count = 1

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

You should run this code to see what happens 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

Higher Order Functions

3.1 Intro

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

3.2 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 sugar⁴ 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 looked liked any of the following

```java
int area(int length, int width){
    return length * width;
}
```

```c
int max(int* arr, int arr_length){
    int max = arr[0]
    for(int i =1; i < arr_length; i++)
        if(arr[i] > max
            max = arr[i];
    return max;
}
```

```python
def str_len(str)
    return len(str)
```

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.

⁴syntactic sugar just means that the syntax looks pretty or sweet like sugar
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.

### 3.3 Higher Order Programming

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

Let us consider the following C program:

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

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

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

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

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.

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

If we consider what is going on in the machine (Maybe recall from 216), then we know that any piece of data is just 1s and 0s stored at some memory address. The variable name helps us know which memory address we are storing things (so we don’t have to remember what we stored at address 0x0012f 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.

### 3.4 Anonymous Functions

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

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

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

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

Which means the following is just syntactic sugar.

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

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

### 3.5 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:

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

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

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

Here `plus2` is a function that takes in an `int` but then returns a function that itself takes in an `int` and returns an `int`. Which means we can actually define `plus` as
1. **def plus():**
2. 
3.    `lambda x: lambda y: x + y`
4. 
5.    `plus(2)(3)`

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

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

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

It is important to note here that you can only partially apply variables in the order used in the function declaration. That is a function like 

```
add = lambda x: lambda y: x + y
```

can only partially apply the `x` variable: `add4 = add(4)`. This is because we are technically doing something like `add4 = lambda y: 4 + y`.

To be more clear:

1. **def sub(x,y):**
2.  `return x - y`
3. # same as let sub = lambda x : lambda y: x - y
4. **def minus3(y):**
5.  `return sub(y,3)`
6. # minus3 = fun y -> y - 3

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

### 3.6 Closures

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

```
def and4(w,x,y,z):
    return w and x and y and z
# and4 = lambda w: lambda x: lambda y: lambda z: w and x and y and z
def and3():
    return add4(True)
# and3 = lambda x: lambda y: lambda z: True and y and z
    return add3(True)
# and2 = lambda y: lambda z: True and True and y and z
```

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

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

It is important to note 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:

1. `sub = lambda x: lambda y: x - y`
2. `x = 3`
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.

3.7 Map

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

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

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

3.7.1 Map

Let us consider the following functions:

```python
def add1(lst):
    ret = []
    for x in lst:
        ret.append(x + 1)
    return ret
def times2(x):
    ret = []
    for x in lst:
        ret.append(x*2)
    return ret
def isEven(x):
    ret = []
    for x in lst:
        ret.append(x%2 == 0)
    return ret
```

All of these functions aim to iterate through a list and modify each item. This is very common need and so instead of creating the above functions to do so, we may want to use this function called Map. Map will map the items from the input list (the domain) to a list of new item (co-domain). To take the above function and make it more generic, let us see that is the same across all of them:
def common(lst):
    ret = []
    for x in lst:
        ret.append(___)  # Add the correct value
    return ret

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

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

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

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

common([1, 2, 3], lambda x: x + 1)

starting at line 2 we set the initial value of ret
ret = []
then we iterate over the list of [1, 2, 3]
iteration 1:
x = 1
ret = []
We then added an item to ret: ret.append(f(x))
f(x) is the same as (lambda x: x + 1)(1)
so ret.append(2) or since ret is [], [].append(2)
so ret is now [2]
iteration 2:
x = 2
ret = [2]
We then added an item to ret: ret.append(f(x))
f(x) is the same as (lambda x: x + 1)(2)
so ret.append(3) or since ret is [2], [2].append(3)
so ret is now [2, 3]
iteration 3:
x = 3
ret = [2, 3]
We then added an item to ret: ret.append(f(x))

\(f(x)\) is the same as \((\lambda x: x + 1)(3)\)

so ret.append(4) or since ret is [2,3], [2,3].append(4)

so ret is now [2,3,4]

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

There are other common or useful HOF that exist, but we will talk about them when we do OCaml.
Chapter 4

Regular Expressions

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

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.

4.1 Introduction

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

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

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

Many would think that if you are trying to read certain data from a string, that methods that deal with strings is the solution. If I loaded a configuration file and wanted to know if I had to set a value to ‘true’ then I would probably want to check if the string has a substring 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.

4.2 Regular Expression Basics

At a basic level, a Regular Expression \(^2\) 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.

\(^1\) for the most part. There is always the Cold Boot Attack

\(^2\)
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.

### 4.3 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|1|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", "0", "000000009" 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 \(\geq 0\). (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 parentheses 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 ^ operator or the end of the string with the $ operator. Consider the following:

```ruby
1  # regular_expressions.rb
2  if /\^where\^/ =~ "anywhere in the string" then
3     puts "matched at the begining"
4  end
5  if /\where\$/ =~ "anywhere in the string" then
6     puts "matched at the end"
7  end
8  if /\where\^/ =~ "anywhere in the string" then
9     puts "matched somewhere in the string"
10 end
```

Here the only thing printed is Matched somewhere in the string.
CHAPTER 4. REGULAR EXPRESSIONS

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 (303) 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)-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$$ refers 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 capture 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: /\([0-9]\)3\([0-9]\)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 reason as to why I do not like this method of matching patterns despite how easy it is.

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

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

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

\(^2\)This is the university’s phone number
### 4.4 Regular Expression In OCaml

//TODO
Chapter 5

Regular Expressions

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

5.1 Introduction

For those that do not know, programs live in RAM on the machine. Since RAM is wiped everytime you power down your machine¹, programs and program memory is designed to live short term. For long term storage, we need to use hard drives, since what is written to a hard drive is saved for a much longer period of time. When we write to the hard drive we typically do this by writing a file and saving it.

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

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

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

5.2 Regular Expression as a Tool

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

¹for the most part. There is always the Cold Boot Attack
A finite state machine will be covered in a later chapter as well. For now however, we can think of regex as a tool (or library) used to search (and extract!) text.

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

- **An Alphabet** - An Alphabet is the set of symbols or characters allowed in the string. If our set of strings are for English words, we would have a different alphabet than one that describes a set of mandarin strings.

- **Concatenation** - Since most strings are longer than a single character, we will need some way to demonstrate the concatenation of single characters to create longer strings.

- **Alternation** - Being able to say one thing or another. We could say that any non-empty set is a union of 2 other sets. So I want to say that a string in set $S$ is in either $S_1$ or $S_2$ where $\{S_1, S_2\}$ is a partition of $S$.

- **Quantification** - The thing about patterns is that repeat. So if I want to have a pattern, then I need to allow for repetition.

- **Precedence** - Ultimately not something that is the basis of regex, but is helpful making sure we know what exactly is being described.

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

### 5.3 Regular Expression Creation

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

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

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

```ruby
' a '
```

That’s it. That’s our very first pattern. What does it mean?

This is a very boring regular expression, it just describes a set with a single element of the “a” string: \{"a"\}.

What about something longer? Maybe my name?

```ruby
' cliff ' 
```

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

```ruby
' hello|hi|good morning|sup|salutations ' 
```

Now we are getting somewhere. We now have something that describes the set: \{"hello", "hi", "good morning", "sup", "salutations"\}. Additionally, we have just seen our first regex operator: the or operation, denoted with a pipebar (`|`).

### 5.4 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: `/`.
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.

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 "abcf" 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", "o", "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 \(r \geq 0\). (eg. "0", "1", "10", "123", "54234543" would all be matched).
  - Exact or bounded repeats: Represented as \{x\}, \{x,y\}, \{x,\} or \cheduler\{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 parentheses 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 two 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]{2}/ would match on both "330" and "310".

• 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 two 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 ^ 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) 405-1000². 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 refers 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 capture group and so on. I am unsure how many capture groups can exist with this method.

²This is the university’s phone number
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}/.

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 reason as to why I do not like this method of matching patterns despite how easy it is.

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

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

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

# what if the pattern and test strings are as follows:
```
pattern = /a[A-Za-z]*/?  
strs = ["a", "abd", "bad"]
pattern = /a[A-Za-z])?$/  
strs = ["a", "abd", "bad"]
pattern = /a+b+c+d*/, # how is this different from /[a-d]/?  
strs = ["abcd", "bad", "cad", "aaaaacd", "bbbbddd"]
pattern = /^(..)$/  
strs = ["even", "odd", "four", "three", "five"]
```

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

5.5 Regular Expression In OCaml

// TODO
Chapter 6

OCaml

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

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.

6.1 Introduction

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

```ocaml
(* 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.

6.2 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 linguistic'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 definition, 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.

6.2.1 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 are even divisible by 2 from the list
+ Return those values

Notice that the imperative instructions tell you how to do something and does so in steps. The declarative instructions tells you what you are looking for and assumes you can just figure out how to do it. If we translate the imperative code to 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 yet¹, but let's look at a declarative 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.

---

¹If you wanted a solution here is one:

```ocaml
let rec even lst = match lst with []-> []|h::t -> if h mod 2 == 0 then h::(even t) else (even t) in even lst
```
### 6.2.2 Side effects and Immutability

One thing that functional programming aims to do is to minimize this idea of a side effect. 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:

\[
f(x) + f(x) + f(x) = 3 \times f(x)
\]

However, if we run the code above, then \( f(x) + f(x) + f(x) = 1 + 2 + 3 \) and \( 3 \times f(x) = 3 \times 1 \). 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?"

### 6.2.3 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 \( \text{false} \). 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 \( \text{int} \). 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:

```ocaml
(* 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:

```ocaml
(* 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.
1 (* letExpression.ml *)
2 let x = 3 + 4 in x + 1
3 (* syntax *)
4 (* 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. Its type is dependent on what the second expression's type is.
1 (let variable = e1:t1 in e2:t2):
t2

We can of course nest these, and since data is immutable in OCaml, variables are overshadowed.
1 (* scoping.ml *)
2 let x = 3 in let y = 4 in x + y (* 7 *)
3 let x = 3 in let x = 4 in x (* 4 *)
4 let x = 3 in let z = 4 + x in let x = 1 in x + z (* 8 *)
5 (* implicit parenthesis *)
6 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:
1 let x = if true then false else true in let y = 3 + 4 in if
   x then y else z
2 (* implicit parenthesis *)
3 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 $p \lor q$. 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 $p \lor q$.

### 6.2.4 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 $e : t$ meaning 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.

1 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 $e1$, $e2$, and $e3$ as long as their types follow the
above rules all the following are valid expressions:

1 if true then 3 else 4
2 if true then false else true
3 if 3 < 4 then 5 + 6 else 7 + 8
4 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 bools are `true` and `false` or expressions that evaluate to `true` and
`false`. So `if 3 then 4 else 5` would be invalid. This idea of substituting any expression with the expected type can be
used for any expression that has ‘subexpressions’. So the following are valid with let bindings and let expressions:

1 let x = if true then false else true
2 let y = 3 + 4 - 10 in if true then y else y + 10
6.2.5 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:

```plaintext
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 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:

```plaintext
(* 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 *)
```

Unlike let bindings a function definition by itself is an expression. The type of a function is represented as a list of types that looks like t1 -> t2 -> ... -> tx -> ty. For example let area l w = l * w has type int -> int -> int. The last type in this list is always the return type, where the preceding types are the types for input.

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

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

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

6.2.6 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 bools, we can only use things like +, -, *, / on ints. If you wanted to do operations on floats then you need to use different operators. See the following:

```plaintext
(* 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.

```ocaml
1 (* compare.ml *)
2 2 > 4 (* false *)
3 "hello" <= "world" (* true *)
4 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 variable and anything after the first = sign is the expression. Anyway, this is important because then what type is inferred from a function like:

```ocaml
1 (* typeInference0.ml *)
2 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:

```ocaml
1 (* typeInference1.ml *)
2 let f x y = 3
3 (* this is equivalent to something like
4 def f(x,y)
5 3
6 end
7 *)
```

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.

```ocaml
1 (* typeInference2.ml *)
2 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 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 -> 'c (we put any function’s type in parenthesis to show it’s a function). Lastly we know that the returned value of apply is whatever f x y returns. Since we know that f returns some value of type ‘c, we can say apply’s return type is also ‘c. Thus the entire type of apply is ('a -> 'b -> 'c) -> 'a -> 'b -> 'c

### 6.3 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.

#### 6.3.1 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:

```ocaml
1 (* list.ml*)
2 [1;2;3;4] (* type: int list *)
```
6.3. OCAML PATTERN MATCHING

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

```
let f1 x = x + 1 in let f2 y = x + 1 in let f3 z = z + 1 in [f1;f2;f3] (* (int -> int) list *)
```

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

### 6.3.2 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 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 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] *)
(*)
(* el1 :: el2 :: t list *)
```

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

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

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

### 6.3.3 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
```
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:

```ocaml
switch (five.osf) {
    case (zero.osf) : return zero.osf;
    case (one.osf ) : return one.osf ;
    case (three.osf) : return two.osf ;
    case (five.osf) : return three.osf ;
    default : return four.osf ;
}
```

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 two.osf items. We can use each of these in a math expression and they mean the same thing.

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

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 four.osf if longer than three.osf elements.

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

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

### 6.3.4 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 *)

```ocaml
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.

```ocaml
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 6.4).

### 6.4 Data Types and Syntax

#### 6.4.1 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.

```ocaml
[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:

```ocaml
(* 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.
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.

```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.

```ocaml
(* variants.ml *)

type coin = HEADS | TAILS
let flip c = match c with HEADS -> TAILS | TAILS -> HEADS

(* flip is a function of type coin -> coin *)
```

```ocaml
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:

```ocaml
(* variants.ml *)

type shape = Rect of int * int | Circle of float

let r = Rect 3 4 (* type is shape, value is Rect(3,4) *)
let c = Circle 4.0 (* type is shape, value is Circle(4.0) *)

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

```ocaml
(* 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.

```ocaml
(* variants.ml *)

type int_tree = Node of int * int_tree * int_tree | Leaf

let t = Node(4,Node(3,Node(2,Leaf,Leaf),Leaf),Node(5,Leaf,Leaf))

(* tree that looks like
  4
 / \ 3 5
 / 2
 *)
6.4.2 Syntax
Chapter 7

Higher Order Functions

7.1 Intro

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

7.2 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 *)

```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.

## 7.3 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;
}

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

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.

```ocaml
let x = 3; // C, Java
y = 4 # Ruby
let z = 4.2;; (* OCaml *)
// idea
```
If we consider what is going on in the machine (Maybe recall from two.osf/one.osf/six.osf), 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 x0o12f 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;;
```

### 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 = 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?

### 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:
CHAPTER 7. HIGHER ORDER FUNCTIONS

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

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

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

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

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

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

It is important to note here that you can only partially apply variables in the order used in the function declaration. That is a function like let add = fun x -> fun y -> x + y can only partially apply the x variable: let add4 = add 4.

This is because we are technically doing something like let add4 = fun y -> 4 + y. If we wanted to partially apply the second parameter we would need to do something let flip f x y = f y x in flip sub.

To be more clear:

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

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

7.6 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:
\begin{verbatim}
let sub x y = x - y
let sub3 = sub 3
(* sub3 may look like
(function: fun y -> x - y, environment: [x:3])
*)
\end{verbatim}

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:

\begin{verbatim}
1 let sub x y = x - y
2 let x = 3
3 let sub3 = sub x
4 let x = 5
5 sub3 5 (* evaluates to -2 since 3 - 5 = -1 *)
\end{verbatim}

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

### 7.7 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:

\begin{verbatim}
1 let sub x y = x - y
2 let div x y = x / y
3 let mystery x y = (x*2)+(y*3)
4 let sub3 y = sub y 3
5 let div3 y = div y 3
6 let double y = mystery y 0
\end{verbatim}

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.

\begin{verbatim}
let flip f x y = f y x
let sub3 = flip sub 3
let div3 = flip div 3
let double = flip mystery 0
\end{verbatim}

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

### 7.7.1 Map

Let us consider the following functions:

\begin{verbatim}
let rec double_items lst = match lst with
1 [] -> []
2 |h::t -> (h*2)::(double_items t)

let rec is_even lst = match lst with
3 [] ->[]
4 |h::t -> (if h mod 2 = 0 then true else false)::(is_even t)

let rec neg lst = match lst with
5 [] -> []
\end{verbatim}
|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:

```ocaml
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

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

Fun fact: map actually exists in Ruby ([1,2,3].map{|x| x+1}). Either way, in OCaml and other languages without imperative looping structures, this is a common recursive function that is needed and can be used to modify each item of a list by building a new list of the modified values (recall that everything in OCaml is immutable). Consider the code trace for adding 1 to each item in a int list.

```ocaml
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 [1])
map add1 [] = []
map add1 [1;2;3] = (add1 1)::(add1 2)::(add1 3)::[]
map add1 [1;2;3] = [2;3;4]
*)
```

### 7.7.2 Fold

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

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

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

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

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

```ocaml
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.

```ocaml
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.

```ocaml
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 operation (recursive_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 `length`? 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:

```ocaml
let myfunc h rc = 1 + rc in
fold_r myfunc [1;2;3] 0
```

```
(* fold_r myfunc [1;2;3] 0 = myfunc 1 (fold_r myfunc [2;3] 0)
fold_r myfunc [2;3] 0 = myfunc 2 (fold_r myfunc [3] 0)
fold_r myfunc [3] 0 = myfunc 3 (fold_r myfunc [] 0)
fold_r myfunc [] 0 = 0
fold_r myfunc [3] 0 = myfunc 3 0 = 1 + 0 = 1
fold_r myfunc [2;3] 0 = myfunc 2 1 = 1 + 1 = 2
fold_r myfunc [1;2;3] 0 = myfunc 1 3 = 1 + 2 = 3
*)
```

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

**fold_left**

See the next section (section 7.8) about why this we can minimize the number of stack frame, but since you already know how fold works, here is fold_left.

```ocaml
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 o [1;2;3]
(*
fold add o [1;2;3] = fold add (add o 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 (-) o [1;2;3]
(*
fold (-) o [1;2;3] = fold (-)((-) o 1) [2;3] = fold (-) -1 [2;3]
fold (-) -3 [3] = fold add((-) -3 3) [] = fold add -6 []
fold (-) -6 [] = -6
*)

(* compare this to fold_right *)
fold_r (-) [1;2;3] o
(*
fold_r (-) [1;2;3] o = (-) 1 (fold_r (-) [2;3] o)
fold_r (-) [2;3] o = (-) 2 (fold_r (-) [3] o)
fold_r (-) [3] o = (-) 3 (fold_r (-) [] o)
fold_r (-) [] o = 0
fold_r (-) [3] o = (-) 3 0 = 3
fold_r (-) [2;3] o = (-) 2 3 = -1
fold_r (-) [1;2;3] o = (-) 1 -1 = 2
*)

(* -6 != 2 *)

How interesting.

### 7.8 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 chapter 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
}

int fact3(int x){
    if (x == 3)
        return 3 * fact2(x-1);
    return -1
}

int fact4(int x){
    if (x == 4)
```
Suppose we are on line 18. To evaluate what is returned, we have to call fact3, wait for its return value, and then use that return value by multiplying it by 4. This is no different than its recursive equivalent

```c
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 its return value, then using its return value, we are instead calling ourselves, 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)

```c
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 our base case, we can then start popping values off. So popping off the text `fact4(1)` call would make the stack look like

```c
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 the 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 stack overflow error.

```c
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.

```c
//Bottom of Stack/
// calling fact4(3)
-----------------------------
4 return 3 * fact4(2)
5 //cannot return here since we need to first calculate fact4(2)
6 -----------------------------
7 return 2 * fact4(1)
8 //cannot return here since we need to first calculate fact4(1)
9 -----------------------------
10 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:

```c
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:

```c
//Bottom of Stack/
// calling fact(3,1)
-----------------------------
4 // fact(3,1)
5 return fact(3-1,3*1) // fact(2,3)
6 -----------------------------
7 // fact(2,3)
8 return fact(2-1,2*3) // fact(1,6)
9 -----------------------------
10 // fact(1,6)
11 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.

```c
//Bottom of Stack/
// calling fact(3,1)
-----------------------------
4 // fact(3,1)
5 return fact(3-1,3*1) // fact(2,3)
```
So why is this called a tail call optimization and how do we make sure we are tail recursive? To answer this question let us look at the syntax of these recursive calls.

```ocaml
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.

```ocaml
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 *)

12
13 let x= 3 * 4 in x + 4
14 (* the last thing here is x+4 so the expression x + 4 is in tail position *)
15
16 let x = 3 + 4 in let x = 6 in 7
17 (* consider the syntax we used for a let binding: let v = e1:t1 in e2:t.
18 Here x = v, e1 = 3 + 4, and (let x = 6 in 7) is e2. Here at the top level, (or in broadest context), the expression in tail position is e2 or (let x = 6 in 7). If we changed our context to be more "zoomed in" or "jump in instead of jump over" then things in tail position would be just 7 *)

Again this is purely a syntactical thing which depends on the context of which parts of the expression will we consider. In an earlier section, we talked about fold_right and fold_left. They both do the same thing(ish), but one of them is tail recursive, and the other is not.
Chapter 8

Finite State Machines

8.1 Introduction

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

8.1.1 Compilers

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

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

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

8.1.2 Background - Automata Theory

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

\(^1\text{See Appendix A for a rough implementation //TODO}\)
A circuit or logic gate is the most basic form of this. If our input is values of true and false (1 and 0), let us put those inputs into a machine that 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. As formalized in the Church-Turing thesis, any solvable problem can be converted in a Turing Machine. A Turing machine that creates or simulates other Turing machines is called an Universal Turing Machine (UTM). Fun fact: Our machines we call computers are UTMs.

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

### 8.1.3 Finite State Machines

Let’s start by modeling a universe and breaking it down to a series of discrete states and actions. Let us suppose that my universe is very small. There is just me, 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 $90^\circ$ and face East, or turn left $90^\circ$ 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:

![Finite State Machine Graph]

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.

---

2Initially called an ‘a-machine’ or atomic machine by Alan Turing.
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, right, right, I can traverse my path (N -> W -> S -> W -> S -> E -> S -> W) to know where I am and return it (My output is W here).

8.2 Regex

So we did this whole thing with graphs and talked about what a machine is and a single example 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 (s0)
- a set of final (or accepting) states (F)
- a set of transitions (δ)

To be clear on types: s0 ∈ S and F ⊆ 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 next second of the string (since we consumed the first one): “0” and consume it. Since we are in state S1, if we see a “0”, then we move to state S0. We then proceed to traverse the graph in this manner until we have consumed the entire string. The traversal should look something like

S0 -0-> 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\(^2\). 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).

\(^2\)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
8.3 Deterministic Finite Automata

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

![Deterministic Finite Automata Diagram](image)

Now this graph is missing a few states (one really). What happens when I am in state \(S_0\) 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 \(S_1\) to the garbage state on the symbol “b”. There are also transitions to the garbage state from \(S_2\) on “a” and \(S_3\) 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):

![Nondeterministic Finite Automata Diagram](image)

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.

8.4 Nondeterministic Finite Automata

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

\(^4\)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
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 \( \epsilon \)-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 \( \epsilon \)-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 - \( \epsilon \) -> 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/.
8.5 Regex to NFA

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

\[
R \rightarrow \emptyset \\
| \epsilon \\
| \sigma \\
| R_1 R_2 \\
| R_1 | R_2 \\
| R_1^*
\]

All this says is that any Regular expression is either

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

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

8.5.1 Base Cases

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

The \(\emptyset\)

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

```
S0
```

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

The empty string (\(\epsilon\))

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:

```
S0
```

Even if \(\Sigma\) 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 $\sigma$

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 $\sigma$. This machine can be represented in the following manner:

![Diagram of a single character NFA]

8.5.2 Concatenation ($R_1 R_2$)

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

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

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

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:

![Diagram of concatenation of two machines]

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:

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

### 8.5.3 Branching \((R_1 | R_2)\)

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

\[
\begin{align*}
R_1 & \quad \text{S0} \xrightarrow{\epsilon} \text{S1} \\
R_2 & \quad \text{S2} \xrightarrow{\epsilon} \text{S3}
\end{align*}
\]

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 \(R_2\), then we are trying to describe \(L_3\) which can be represented as \(\{x \mid x \in L_1 \lor x \in L_2\}\). 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 one start state. The easiest way to do so is by making use of \(\epsilon\)-transitions with two new states. Here is the resulting machine:

\[
\begin{align*}
\text{S0} & \xrightarrow{\epsilon} \text{S1} & \text{S2} & \xrightarrow{\epsilon} \text{S3} & \text{S4} & \xrightarrow{\epsilon} \text{S5} & \text{S4} & \xrightarrow{\epsilon} \text{S5}
\end{align*}
\]

This new machine still has one starting state, and one accepting state which means we can inductively build larger machines, and the \(\epsilon\)-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:

\[
\begin{align*}
\text{S0} & \xrightarrow{\epsilon} \text{S1} \quad \text{S0} & \xrightarrow{a} \text{S1} & \text{S2} & \xrightarrow{\epsilon} \text{S3} \quad \text{S2} & \xrightarrow{b} \text{S3} & \text{S4} & \xrightarrow{\epsilon} \text{S5} \quad \text{S4} & \xrightarrow{\epsilon} \text{S5}
\end{align*}
\]

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

### 8.5.4 Kleene Closure \((R_1^*)\)

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

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

The Kleene Closure of a language, is just analogous to the language’s regular expression with the \( * \) operator. That is, if \( L_1 \) is the language corresponding to \( R_1 \) then we are trying to describe \( L_2 \) which can be represented as \( \{ x | x = \epsilon \lor x \in L_1 \lor x \in L_1 L_1 \lor x \in L_1 L_1 \lor \ldots \} \). 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 \( \epsilon \)-transitions become really important. To accept the empty string, we just use an \( \epsilon \)-transition to move from \( S_2 \) to \( S_3 \). For repeated values, we can just use the \( \epsilon \)-transitions from \( S_3 \) to \( S_2 \). 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

\[ S_2 \xrightarrow{\epsilon} S_3 \]

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

\[ S_2 \xrightarrow{\epsilon} S_0 \xrightarrow{a} S_1 \xrightarrow{\epsilon} S_3 \]

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

\[ S_2 \xrightarrow{\epsilon} S_0 \xrightarrow{a} S_1 \xrightarrow{\epsilon} S_3 \xrightarrow{\epsilon} S_2 \xrightarrow{\epsilon} S_0 \xrightarrow{a} S_1 \xrightarrow{\epsilon} S_3 \]

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

### 8.5.5 Example

For a quick example, if we wanted to make the NFA for the regular expression: \( / (ab|cd)+/ \) the machine (with a few optimizations due to space) would look like:
We could optimize this even further, but not really needed. Additionally to see without any optimizations and for a step-by-step, you can see Appendix B (/TODO).

### 8.6 NFA to DFA

Now that we know how to convert from a regular expression to a NFA, we should talk about 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:

- S0
- S1
- S2
- S3

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).

```plaintext
S0 \rightarrow a \rightarrow S1 \rightarrow b \rightarrow S2 \rightarrow b \rightarrow S3
```

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 \rightarrow a \rightarrow S1 \rightarrow Garbage
- S0 \rightarrow a \rightarrow S0 \rightarrow a \rightarrow S0 \rightarrow b \rightarrow S0 \rightarrow b \rightarrow S0
- S0 \rightarrow a \rightarrow S0 \rightarrow a \rightarrow S1 \rightarrow b \rightarrow S2 \rightarrow b \rightarrow 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 \rightarrow a \rightarrow S0 or I could go from S0 \rightarrow a \rightarrow 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.

```plaintext
S0 \rightarrow a \rightarrow S0, 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 \( S_0, S_1 \), 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. So going back to the original NFA, if I am in state \( S_0 \) and see a "a", I could go to state \( S_0 \) or \( S_1 \). Additionally, (looking at the original NFA), if I am in state \( S_1 \) 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 \( S_0 \) or \( S_1 \), if I see an "a", then I have to either be in state \( S_0 \) or \( S_1 \). Well we already have a state that represents this possibility so we can just add the following transition:

\[
S_0 \rightarrow \{ S_0, S_1 \}, S_1 \rightarrow \emptyset, S_0 \lor S_1 \rightarrow \{ S_0, S_1 \}
\]

If this is confusing, consider the following logical argument:

\[
\begin{align*}
p & \Rightarrow q \\
s & \Rightarrow r \\
p \lor s
\end{align*}
\]

\[
\therefore q \lor r
\]

From CMSC250 we know this is a valid logical argument (known as constructive dilemma). The same applies here. If \( S_0 \) and "a" leads to \( S_0 \) and \( S_1 \), and \( S_1 \) and "a" leads nowhere (except the garbage state) then we will and up in either \( S_0 \) or \( S_1 \) (or the garbage state).

Anyway, that was a slight sidetrack. We next need to consider if we are in \( S_0, S_1 \) and we see a "b". The above logic applies. If I am in state \( S_0 \) and see a b (looking at the original NFA), then I will end up in state \( S_0 \). If I am in state \( S_1 \) of the original NFA and see a b, then I will end up in state \( S_2 \). It is uncertain which state I will be in though so let's add a new state that represents this possibility so we can just add the following transition:

\[
S_0 \rightarrow \{ S_0, S_1 \}, S_1 \rightarrow \emptyset, S_0 \lor S_1
\]

In "laymen's" terms, we are in quantum superposition, that is we are in both \( S_0 \) and \( S_1 \) at the same time.

Now if I am in state \( S_0 \) and see a "b" then I can only go to state \( S_0 \). If I am in state \( S_2 \) and see a "b", then I can only go to state \( S_3 \). So if I am in either state \( S_0 \) or \( S_2 \), then I will end up in either \( S_0 \) or \( S_3 \). Here is another place of uncertainly so let us add this new state with transition.

\[
S_0 \rightarrow \{ S_0, S_1 \}, S_1 \rightarrow \emptyset, S_0 \lor S_1
\]
Again, the same issue arises. If I am in state $S_0, S_3$, 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 $S_0$ and see a “a”, then I could either be in $S_0$ or $S_1$. If I am in state $S_3$ and see a “a”, then I can go nowhere (except a trash state). So if I am in either state $S_0$ or $S_1$, then I can only end up in $S_0$ or $S_1$. Let us add this transition.

Now if I am in state $S_0$ and see a “b” then I can only go to state $S_0$. If I am in state $S_3$ and see a “b”, then I cannot go anywhere (except garbage). So if I am in state $S_0$ or $S_3$, if I see a “b”, I can only really go to $S_0$. 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 $\epsilon$-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, $S_3$ 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 $S_3$ as an accepting state.

If you also go back a few pages, this machine is identical to the DFA we said corresponded to our NFA. Wild.
8.6. NFA TO DFA

8.6.1 NFA To DFA Algorithm

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

To do so, we will need to define 2 subroutines: \( \epsilon \)-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:

- \( \Sigma \): 'a list, a list of symbols
- \( S \): 'b list, a list of states
- \( s_0 \): 'b, a single state \( (s_0 \in S) \)
- \( F \): 'b list, a list of state we should accept \( (F \subseteq S) \)
- \( \delta \): (‘b * ‘a * ‘b) list, a list of transitions from one state to another, ((source, symbol, destination)).

\( \epsilon \)-Closure

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

If I am in state \( S_0 \), I could also possibly be in state \( S_1 \) as well. So \( \epsilon \)-closure if a function that helps us figure out where can we go using only \( \epsilon \)-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 \( \epsilon \)-transitions. It is important to note that any state can reach itself via an \( \epsilon \)-transition. So here is the type of \( \epsilon \)-closure:

- \( \epsilon \)-closure: \((\text{NFA} \rightarrow \text{'b list} \rightarrow \text{'b list})\), given a list of states, return a list of states reachable using only \( \epsilon \)-transitions

To see an example, let us consider the following machine:

\[
S_0 \xrightarrow{\epsilon} S_1
\]

If I were to call \( \epsilon \)-closure \( \text{nfa [S0]} \) I should get back \([S0;S1;S2]\). The best way to do so is by iterating through \( \delta \) and checking where you can go to anywhere in the input list. Then recursively calling \( \epsilon \)-closure on the resulting list. That is:

\[
e-\text{closure nfa [S0]}\\
// Looking at S0 I can only go to S0 and S1 via an epsilon transition [S0;S1]\\
// looking at S0 I can go to S0 and S1, looking at S1 I can go to S1 and S2 [S0;S1;S2]\\
// looking at S0 I can go to S0 and S1, looking at S1 I can do to S1 and S2, [S0;S1;S2]\]
// 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 e-closure at any point during a move. For an example, consider the following machine:

![Diagram of a state transition graph]

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

8.7 DFA to Regex
Chapter 9

Languages

9.1 Introduction

We have now done a ton in regex but the issue with regular expressions is that they can only do so much. Regular expressions are great when talking about regular languages, but programming languages (and spoken languages) are not regular languages and so we need something more expressive. So let us consider what exactly we need to describe a language.

9.2 Context-Free Grammars

Recall that a language is just a set of strings. We used regular expressions to tell you how to construct the strings in the set. An alternative way is to give a recursive definition of the set. Recall from five what a recursive set is (in this case the set of positive multiples of 3):

$$S = \begin{cases} 3 \\ x \in S \Rightarrow x + 3 \in S \end{cases}$$

We can do the same thing to describe sets of strings (although we use a slightly different notation). This is called a grammar. We will be going over context-free grammars (CFGs) as opposed to context-sensitive grammars (CSGs). For example let us take the CFG we saw for regular expressions:

Any regular expression can be expression as a string that is in the following set

$$S \rightarrow \epsilon \\
| \sigma \\
| SS \\
| S|S \\
| S^* \\
| (S)$$

That is, any valid regular expression sentence follows the above pattern. Let's break this down and understand exactly what it means. Here S is called a Non-terminal because S could be a variety of things. On the other hand symbols like ε, σ, *, |, (, ) are what we call terminals. Grammars also have what is called productions: rules about what non-terminals can be. This example is honestly not the best for these terms but we will see an example soon and we will revisit these terms.

The important part right now is that this grammar describes all strings that represent a regular expression. Very much like we can use a FA or to trace through a regular expression to show that the regular expression accepts a string, we can
derive a string from a grammar using substitution to show that a string is grammatically correct (and hence belongs in the language the grammar describes).

For example, the regular expressions /ab*/ can be described using the following derivation:

\[
S \rightarrow SS \\
\quad \rightarrow aS \\
\quad \rightarrow aS^* \\
\quad \rightarrow ab^*
\]

Now for those that have taken a linguistics or hearing and speech sciences class, or even an English class know that grammar typically refers to the order in which words need to be to make sense. That is still true here. The regular expressions /+b/ cannot be derived from the above grammar so we claim it to be grammatically incorrect and not part of the language.

Now English is complicated and has a lot of rules but consider the following simplified grammar for English.

\[
S \rightarrow NP \ VP \\
NP \rightarrow \text{pronoun} \ |
\text{proper\_noun} \ |
\text{det AN} \\
AN \rightarrow \text{adj AN} \ |
\text{noun} \\
VP \rightarrow \text{verb} \ |
\text{verb NP}
\]

Let us revisit our terms to break this down. Non-terminals are symbols that represent other symbols. Conventionally we give them uppercase letters. Sometimes the letters mean something, sometimes they are just alphabetical. In this case they mean something.

- **Terminals**: pronoun, proper\_noun, det, adj, noun, verb are all terminals. Unlike the previous example where a lowercase letter was a symbol, these all stand for larger sets of things (This can be confusing, so in this course we will typically only be using symbols like in the first example).

- **Non-terminals** $S$, $NP$, $VP$, $AN$ are all non-terminals and we know this because they are all uppercase, and each have a production rule associated with them.

- **Production**: a production rule tells us all the things a non-terminal can be. For example $S \rightarrow NP \ VP$ is a production rule. It states that any sentence $S$ consists of a noun phrase $NP$ followed by a verb phrase $VP$.

$AN \rightarrow \text{adj AN}|\text{noun}$ says that any $AN$ phrase is an adjective followed by another $AN$ phrase or is just a noun.

Using this grammar we can still derive if a sentence is a grammatically valid one in English. For example, if I had a sentence like "The child ran the race" then I could say, that this sentence is grammatically correct and should be in the set of valid English sentences. Before I show the derivation let us make sure we know our parts of speech:

- "The" is a determiner (det) because it determines the reference of something. Some other examples include are "every", "a", "some", or "each".
- "child" and "race" are nouns since they fall under the category of person, place, or thing (or idea).
- "ran" is a verb since it describes an action.

Knowing all this let us show the derivation:

\[
S \rightarrow NP \ VP \\
\quad \rightarrow \text{The (det) AN VP} \\
\quad \rightarrow \text{The child (noun) VP} \\
\quad \rightarrow \text{The child ran (verb) NP} \\
\quad \rightarrow \text{The child ran the AN} \\
\quad \rightarrow \text{The child ran the race}
\]

That is, a sentence $S$ is a noun phrase ($NP$) followed by a verb phrase ($VP$). The first noun phrase is going to use the third definition of a noun phrase: det $AN$. "The" is the determiner. The following $AN$ is using the second definition of being
just a noun, in this case "child". So the noun phrase is "The child". The verb phrase is going to use the second definition of a verb phrase: verb \( NP \). The verb here is "ran" and the noun phrase is going to be "det \( AN \). In this case the determiner is again "the" and the \( AN \) is just a noun: "race". More on this in a bit.

### 9.3 Designing Grammars

We said that a grammar describes a set of strings and is more expressive than regular expressions. This means that any regular expression can also be expressed as a CFG but also that CFGs can get around some restrictions that regular expressions have. Let us start with regular expression supported operations.

#### 9.3.1 Regular Expressions Supported

Let us start with our 3 base cases.

- **∅**: If the language is empty, then the CFG should reflect that as well. The CFG can still be represented as \( ∅ \)

- **ε**: If the regular expression accepts the empty string, then the CFG can just be a single production.

\[
S \rightarrow ε
\]

- **σ**: If the regular expression is just a single character then we can have our CFG reflect that character in a single production.

\[
S \rightarrow σ
\]

I will say though that in larger grammars typically we are describing sentences or statements with words so if we have a list of words we can do the same thing. For example

\[
S \rightarrow \text{Cliff}
\]

Once we have the base cases down we can talk about the recursive definitions: concatenation, branching, and kleene closure:

- **Concatenation**: If we wish to concatenate two things together we can just push them together with either non-terminals or just the string you expect. For example a the corresponding CFG for the regular expressions /ab/ would be

\[
S \rightarrow ab
\]

Alternatively you could do something like

\[
S \rightarrow AB
\]

\[
A \rightarrow a
\]

\[
B \rightarrow b
\]

This can be helpful if you have branching (next bit), or just a sentence where you want to force one thing to come before (eg. adjective before noun).

- **Branching**: If we want to branch, we can use the same symbol we used in regex |. For example, the grammar for a greeting could be

\[
S \rightarrow \text{Hello}|hi
\]

You can put each option on a new line if you have the space, but either way is valid.

- **Kleene Closure**: For allowing repeated values, we can just utilize the recursive property these sets have. For example, the corresponding CFG for the regex /a*/ is

\[
S \rightarrow aSε
\]

. CFGs also let us have a shortcut for something like /a+/ 

\[
S \rightarrow aS|a
\]

We can also use this to repeat whole words:

\[
S \rightarrow \text{This is a } T \text{ sentence}
\]

\[
T \rightarrow \text{very } T|long
\]
9.3.2 Not supported by Regular Expressions

We said that CFGs are more expressive which means we can say more with a CFG than we can with regular expressions. So let us think of some of the restrictions we had with regular expressions. We could not look forward more than one character at a time, and we could never reference what we previously saw. So would couldn’t do things like balanced parenthesis. This is all in thanks to the recursive nature of CFGs.

Consider the previous CFG:

\[ S \rightarrow \text{This is a } T \text{ sentence} \]
\[ T \rightarrow \text{very } T \mid \text{long} \]

We have a sentence where we have something known ("This is a") followed by non-terminal (\( T \)) which is then followed by some other known string ("sentence"). By allowing for this ability to look at both before and after the non-terminal, we can do things like balance parenthesis, or have relative distinct values.

For something like having a balanced values on either side (parenthesis or palindromes), we can just put the values on either side of the non-terminal.

- Balanced parenthesis surrounding "a"
  \[ S \rightarrow (S)a \]
- Palindromes of "a", "b", and "c"
  \[ S \rightarrow aSa|bSb|cSc|\varepsilon \]

For having relative number of values we are a tad restricted to a few characters that are relative to each other, but supposed we want a string with the same number of "a"s followed by the same number of "b"s. Our notation for this is \( a^n b^n \). The following grammar would allow for that:

\[ S \rightarrow aSb|\varepsilon \]

We can also do distinct relative numbering like \( a^n b^{2n} \):

\[ S \rightarrow aSbb|\varepsilon \]

Or even \( a^n b^m \), \( m \geq n \)

\[ S \rightarrow aSb|T \]
\[ T \rightarrow bT|\varepsilon \]

Sometimes the order doesn’t even matter. If I wanted a string that had the same number of "a"s and "b"s in any order then I could do something like:

\[ S \rightarrow aSb|bSa|\varepsilon \]

I can also do a string that has an unequal amount of "a"s and "b"s in any order:

\[ S \rightarrow A|B \]
\[ A \rightarrow CaA|CaC \]
\[ B \rightarrow CbB|CbC \]
\[ C \rightarrow aCbC|bCaC \]

9.3.3 A basic Grammar

Putting all this together, I could create a rudimentary language that describes basic algebraic expressions.

\[ A \rightarrow A + A|A - A|A * A|A/A|A/N \]
\[ N \rightarrow \text{number} \]

This grammar is okay because it allows for strings like "2 + 1 + 0 - 4 * 9 / 3". However, this grammar does not allow for things like "(4 - 3) + 5", which of course is allowing order of operations to be expressed. We can just easily modify this expression by adding our parenthesis rule we talked about:

\[ A \rightarrow A + A|A - A|A * A|A/A|A/(A)|N \]
\[ N \rightarrow \text{number} \]

Now from a compiler/interpreter stand point these this grammar still has some issues, but we will talk about all of this between the next section and the Parsing chapter.
9.4 Modeling Grammars

Now that we know what a CFG is and what they represent, we need to discuss how we can model them into something useful (since typically you need to do something with a language and not just know what they are and how they work).

Now one of the leading theories in linguistics and psychology (that I know of) is that we store grammar and the like as a tree in our heads. Regardless if I am up to date or not in the linguistics field, this is what we will be using to model our grammars and sentences in compsci.

Recall that a grammar tells you the structure of a language, so the tree should tell us this as well. We do this by our recursive definition. Consider our basic English Grammar

\[
S \rightarrow NP \ VP \\
NP \rightarrow \text{pronoun} | \text{proper\_noun} | \text{det\ noun} \\
VP \rightarrow \text{verb} | \text{verb\ NP}
\]

If we take the same sentence we always used: "The child ran the race", we can represent this sentence as a tree thanks to our grammar:

If we consider our basic algebraic expression grammar we can also model sentences with it:

\[
A \rightarrow A + A | A - A | A \cdot A | A / A | (A) | N \\
N \rightarrow \text{number}
\]

For example "2 + 3 * 4" can be modeled as:

Technically this tree represents the following derivation:

\[
A \rightarrow A \cdot A \\
A \rightarrow A + A \cdot A \\
A \rightarrow N + A \cdot A \\
A \rightarrow 2 + A \cdot A \\
A \rightarrow 2 + N \cdot A \\
A \rightarrow 2 + 3 \cdot A \\
A \rightarrow 2 + 3 \cdot N \\
A \rightarrow 2 + 3 \cdot 4 \\
N \rightarrow 2 + 3 \cdot 4
\]

This derivation I got via something we call a "left hand derivation". That is, we will substitute the left most variable for a recursive definition. Consider the right hand derivation for the same tree:

\[
A \rightarrow A \cdot A \\
A \rightarrow A \cdot N \\
A \rightarrow A \cdot 4 \\
A \rightarrow A + A \cdot 4 \\
A \rightarrow A + N \cdot 4 \\
A \rightarrow A + 3 \cdot 4 \\
A \rightarrow N + 3 \cdot 4 \\
A \rightarrow 2 + 3 \cdot 4
\]
Notice that using a left hand or right hand derivation does not change the tree (but if we did more, it would impact the way the tree is built. However, notice we could have used the following left hand derivation instead:

\[
A \rightarrow A + A \\
\rightarrow N + A \\
\rightarrow 2 + A \\
\rightarrow 2 + A * A \\
\rightarrow 2 + N * A \\
\rightarrow 2 + 3 * A \\
\rightarrow 2 + 3 * N \\
\rightarrow 2 + 3 * 4
\]

This tree would look like:

```
A
  +
   N
  2
```

```
A
  *
   N
  3
```

Both of these trees and derivations are both valid when substituting the leftmost variable for a definition of \(A\), so we call this grammar ambiguous. A grammar is ambiguous when there are two valid left hand derivations. A grammar can also be ambiguous when where are 2 valid right hand derivation.

Now that we have some tree model, we need to discuss what we do with these trees. These trees have a proper name as well: parse trees. We will get into parsing in a future chapter, but ultimately should we want to try and obtain meaning from a the tree we need some sort of tree traversal algorithm.

In this case, a post order traversal would probably be helpful here. Consider the second variation to the tree we had. If I wanted to calculate the right-most \(A\), then I would need to figure out what two values my sub children were before I said "3-4". Then I could recursively figure out subtrees until I get to the root. That is I would go from the above tree to the below tree:

```
A
  *
   N
  3
```

```
A
  *
   N
  4
```

Which would then have the two subtrees be evaluated to 2 and 12 respectively, which we would then multiply to get 24 as the final result.

However, consider the first variation of the tree. If we simplified in this manner we would get the following:

```
A
  *
   N
  2
```

```
A
  *
   N
  3
```

Simplified down to:

```
A
  *
   N
  4
```

And finally to:

```
A
  *
   N
  4
```

Notice that we get two separate values due to the ambiguity.

There are two ways to help solve ambiguous grammars: we could try and figure out more the grammar and restrict how we go about traversing through the tree, or we can just change the grammar a bit.
For example, much of the ambiguity is because we don’t know which path the variable should be when given something. We can fix this by adding separate non-terminals:

\[
A \rightarrow N + A|N - A|N * A|N / A|(N)(A)
\]

\[
N \rightarrow \text{number}
\]

Now we know that any expression must start with a number and not an expression. So constructing a tree of “2+3+4” could only result in the following:

The final issue here is that when we do our traversal, we are still doing addition before multiplication. In order to fix this, we can either mandate that all operations be put in parenthesis or we can change the grammar to have precedence.

The idea of precedence is that we do things of higher precedence closest to the base case as possible. That is notice that when multiplication is closest to the bottom of the tree, we are getting the correct computation. So let us modify our grammar to have precedence. We use the same sort of trick we did for concatenation, we add more non-terminals since we need to figure out non-terminals before terminals. We should get:

\[
A \rightarrow B + A|B - A|B
\]

\[
B \rightarrow N * B|N / B|N
\]

\[
N \rightarrow \text{number}|(A)
\]

Given this grammar we can only get the following tree:
Chapter 10

Interpreters and Compilers

10.1 Introduction

We know how a grammar describes a language. Let us consider the following language which I will call Math-ew:

\[ E \rightarrow +NE|−NE|∗NE|/NE|N \]
\[ N \rightarrow \{0|1|2|3|4|5|6|7|8|9\} \]

Math-ew describes simple mathematical expressions. Some sentences in this language include "+ 3 4" and "- 3 * 4 + 5 2". We will use Math-ew throughout this chapter so feel free to refer back to it.

Now that we have a language to work with, the next question is how do we go from something like "+ 3 4" to what we can assume to be the correct value, 7? That is, in utop or i.r.b, why is it that when we enter something like "3 + 4" we get back 7?

This is all the work of an interpreter. I personally call this a compiler, but the connotation is important.

10.2 Compilers/Interpreters

Consider what happens when we take a file like "funs.ml" and then run ocamlc funs.ml. It is ultimately the same thing that happens if we ran something like gcc prog.c or javac Program.java. We take a text file ¹ and compile it down to machine code and then we get some program we can run. Practically though we take the text file and convert it to an assembly file (technically another text file) which an assembler then converts to the appropriate machine code.

All of this is to say we think of compilers as something that takes our code and creates a program, but this is practically incorrect. A compiler is a language translator. So you could theoretically create a Java to C compiler, or a Ruby to Java compiler ².

An interpreter on the other hand takes a text file and returns a value. i.r.b is an interpreter since it does not make assembly or machine code, but instead gives back a value. I would argue this is also a compiler because I could just define the target language as something like

\[ S \rightarrow \text{value} \]

However maybe what I should say is that both interpreters and compilers are translators, but they translate in 2 different ways. A compiler translates by making machine code and converting an entire program to an analogous program in a different language. An interpreter translates by converting one statement at a time and evaluating these statements to a value.

¹The only difference between "funs.ml" and "funs.txt" is the file name (which is arbitrary).
²See https://pandoc.org/
Regardless of if we are talking about an interpreter or a compiler, typically there are three things needed to make a translator:

- lexer: converts text file to a list of tokens
- parser: takes list of tokens and creates an intermediate representation (typically a tree)
- evaluator/generator: takes the intermediate representations and either evaluates to a value, or generates analogous code in a different language.

We will talk about all of these things with an example for Math-ew written in C.

### 10.3 Lexing

When you are reading this, you are looking at squiggly lines made up of ink or pixels and being literate, you are able to make meaning of these squiggly lines. Like what a superpower: you can look at squiggly lines and then gain knowledge from said squiggly lines.

Lexing is analogous to figuring out what the words are. Consider the following sentence:

> Your tongue does not fit comfortably in your mouth.

You see some squiggly lines and then get upset that you are now thinking how you should position your tongue. Magic. But lexing is just the process of figuring out what words are in the sentence (and maybe what type they are). That is, you split up the sentence into nine words: "Your", "tongue", "does", "not", "fit", "comfortably", "in", "your", and "mouth". You may even tag certain words as a noun or verb or determiner. However, notice that we said it's just figuring out what the words are. Given the string

> "green the truck"

You still recognize there are three words, "green", "the", and "truck". Despite this being grammatically incorrect, you still were able to figure out these words. That is exactly what a lexer does.

When creating a new language, you would typically have a list of words which should be allowed in the language. In Math-ew we have operator words ("+", "-", "+", and "/") and digit words ("0", "1", "2", ..., "8", "9"). So we should have something like

```c
#define type token = Plus|Sub|Mult|Div|Num of int
(*
If we wanted, we could do something like
```

We then want a function that takes a string and returns a list of tokens. We could do this by doing something like the following:

```c
let rec lex str =
  if str = "" then [] else
  if String.sub str 0 1 = "+" then
    Plus::(lex (String.sub str 1 ((String.length str) - 1)))
  else if String.sub str 0 1 = "-" then
    Sub::(lex (String.sub str 1 ((String.length str) - 1)))
  else if String.sub str 0 1 = "*" then
    Mul::(lex (String.sub str 1 ((String.length str) - 1)))
  else if String.sub str 0 1 = "/" then
    Div::(lex (String.sub str 1 ((String.length str) - 1)))
  else if String.sub str 0 1 = "0" then
    Num(0)::(lex (String.sub str 1 ((String.length str) - 1)))
  ...
  else if String.sub str 0 1 = "9" then
    Num(9)::(lex (String.sub str 1 ((String.length str) - 1)))
  else lex (String.sub str 1 ((String.length str) - 1))
```
This is not the most efficient way to do this. This also assumes you will be fed valid input (a non-malicious user). I would recommend looking at the \texttt{Str} library which lets you use regular expressions in Ocaml.

However, this does work as a lexer.

\begin{verbatim}
let rec lex str =
  lex "1 + 2" = [Num(1); Plus; Num(2)]
  lex "1 2 +" = [Num(1); Num(2); Plus]
  lex "+ 1 2" = [Plus; Num(1); Num(2)]
\end{verbatim}

Again, notice that our lexer does not check if the string matches the grammar. All it does it check if all the words in the string are valid. In this particular lexer, we skip over any unwanted words but may not always be the desired behavior.

\section{Parsing}

Now that we have a list of tokens and know that we have a list of valid words (tokens), we need to make sure our sentence is grammatically correct. That is, if we have the string

\begin{verbatim}
green the truck
\end{verbatim}

You lex this correctly and recognize all three words ("green", "the", "truck") are valid words in the English language, but you notice this is grammatically incorrect. This is what the parser does. Check if things are grammatically correct or not, while also storing the sentence in an intermediate form, whether it be a tree, code, or something else.

This is typically done to make the evaluator or generating step easier, but from a linguistic point, we gain information just by having something be grammatically correct. Consider the sentence "I like purple" and "I like purple shirts". By being grammatically correct, we know that in the first sentence “purple” is a noun, whereas in the second sentence “purple” is an adjective. It would be harder to identify this if things were not grammatically correct.

Additionally there are many types of parsers that exist so depending on the grammar or what you want to prioritize using a different parser may be useful. I will be using a Left-to-right Left-derivation parser, that looks ahead by 1 token at a time or a LL(1) parser. A LL(1) parser is a type of a Recursive Decent Parser (RDP). This means that we will be using a left most derivation when checking the grammar, reading from left to right. The lookahead by 1 just means we will be looking one token at a time (to decide why production to use if needed).

Hence why we may need to change our grammar to match what our parser is capable of. Regardless of which parser you use, you probably want to have a different data structure of representing the sentence. When talking about CFGs, we used a tree which is what we will use here. We will be using an Abstract Syntax Tree (AST) which will just make a tree based off the content of the language. We could make a parse tree instead which makes a tree based off the symbols of our language, but parse trees focus more on non-terminals (internal nodes) vs terminals (leaves) whereas ASTs focus more on content of the string.

Let us consider the following Math-ew sentence “+ 2 - 3 4". The parse tree would look like:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{parse_tree.png}
\end{figure}

Notice this is more closely aligned with the grammar. Where \emph{E} is either \emph{N} or some operator followed by two other \emph{E} values.

For an AST, we care more about what is occurring. Here is an AST for the same sentence:

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{ast.png}
\end{figure}

Either option is a valid output for a parser, but for more complex grammars, we sometimes don't care about all tokens (like white space or \{\}) or we can represent some tokens use via the structure of the tree. For example if we had parenthesis
to show order of operations, the above AST would not need to include them since the structure of the tree shows that we want to subtract 4 from 3 before we add 2. Hence a parse tree could be replaced with sometime more useful.

So if I wanted to create an AST, then I have to consider how to define and design my tree. The nodes in my tree will either be a number (leaf) or a operation with the subtrees being children of the node. Translating this to code looks like:

```plaintext
type ast = Int of int
  |Add of ast * ast
  |Minus of ast * ast
  |Times of ast * ast
  |Divide of ast * ast
```

Now we need to make the parser. For Math-ew (and typically other recursive languages), we need to consider the fact that for a sentence like "- 2 * 2 3", that the character "2" is a stand alone sentence. However we cannot recursively call here despite the fact that $E \rightarrow N$. That is we run into a small issue:

```plaintext
let rec parser tok_list = match tok_list with
  |[] -> failwith "Math-ew does not support empty strings"
  |Plus::t -> let next_expr = parse t in Add(next_expr, ?)
  ...
```

We have no idea what should be in the spot of the question mark. Since parse t would parse the rest of the list and still have something left over (namely [Mult; Num(2);Num(3)]).

We can solve this problem in 2 ways, either we find some way to figuring out our remaining tokens or we can consider how our language is structured. Let's do the latter since it's harder.

In our language, we know that any operator will be followed by a number or an expression. Hence, we can mimic our grammar with our parser like so:

```plaintext
let rec parse tok_list = match tok_list with
  |Num(x) -> Int(x)
  |Plus::Num(x)::t -> Add(Int(x), parse_E t) (* + N E *)
  ...
```

Here, we can break down each branch as an operator, followed by a number, followed by the remainder. Technically though we are looking ahead by 2 tokens so this is a LL(2) parser. We can easily fix this in the following way:

```plaintext
let rec parse tok_list = match tok_list with
  |Num(x) -> Int(x)
  |Plus::t -> let h::t = t in Add(Int(h),parse t)
  ...
```

A slight difference, but syntactically shows that we are only looking 1 token ahead at a time instead of 2. We can expand on this more in appendix D //TODO.

Notice that this also will only match on grammatically correct sentences. So if our sentence was something like "+ - 2 3 4" we would not match and we would probably throw an error. Which means that now all we have left is to take meaning from the sentence.

### 10.5 Evaluating/Generating

Now that we have a parser that generates a tree of some sort, now we need a way to traverse through the tree to compute a final value (at least for an interpreter). We need to create meaning from our AST. So suppose that we have a sentence like:

```
Colorless green ideas sleep furiously
```

This contains all valid words in the English lexicon (our lexer says ok), and it is grammatically correct (our parser would make a tree) but in English this means nothing. The evaluator's job is to make sure this can be done. We helped this process by designing our AST in a way that we can easily traverse where each node has meaning.

That is if we consider the above AST, notice that we just need to perform a post order traversal to compute a value. That is, at the root, we need to process our left subtree and then our right subtree, and then we can add our values together. This can very easily be modeled with the following code:
let rec eval ast = match ast with
  | Int(x) -> x
  | Add(x,y) -> let left = eval x in
              let right = eval y in
              left + right
  ...

TA-DA! We now have a way to get a single value from a string. Namely:

eval (parse (lex "+ 2 3")) = 5

Typically an interpreter will take a program statement by statement and do this exact thing. A compiler will look at a list of
string (read text file) and computer an analogous text file.

Notice that technically, the meaning from this evaluator is created purely from what we decided Add would do. Addition-
ally this language is very simple. Once we start adding more to our language, it may be possible that we have something that
follows our grammar but makes no sense. Typically we can fix this by changing our parser, or rewriting the grammar (there
are multiple different grammars that express the same language). In some cases, type checking is done during evaluation
(like in ruby).

Suppose that I had a statement that looked like "x+ 1". In most programming languages this is grammatically correct,
variable added to a constant. However if \( x = \text{true} \) then trying to do this operation would fail and it would be meaningless.
Some languages get around this by casting, or by creating a new data type or behavior (see javascript and C).

This idea of defining behavior of a language is a branch of semantics. One particular type called operational semantics
is the next topic.
Chapter 11

Operational Semantics

I am not a fully operational person

11.1 Introduction

Now that we can design a language, we may want to do a few of things, two of which we will talking about here:

• Give meaning to the language

• Prove correctness of a program

Both of these goals can be achieved through the use of operational semantics. Semantics referring to the meaning of a statement, and operational referring to how something operates.

11.2 Meaning

If you ever take a philosophical linguistics course\footnote{Would recommend Phil360: Philosophy of Language with Alexander Williams} 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 \( \text{fun } x \to x \ 3 \) to someone unfamiliar with functional programming?

There’s plenty of ways that you can describe meaning. In programming language theory there are typically three major ways: denotations semantics, axiomatic semantics and operational semantics.

• Denotations: describe meaning via mathematical constructs

• Operational: describe meaning via how something operates

• Axiomatic: describing meanings via axioms

How I think about (and I am sure that people more in both the linguistics and PL space would be mad at me) is that denotational semantics is by giving a definition. For example: "Blue" refers to light waves that fall in-between 450 and 495 nm. Axiomatic semantics gives examples. For example: ‘the sky, the ocean, and that person’s eyes are blue’. Operational semantics describe how we use it. Example: "Blue" is referring to a shade people see between green and violet.

So when we talk about the meaning of a program, we want to talk about it in terms of how the program operates. More specifically, we use operational semantics to communicate language design ideas. If we want to talk about another language
however let's use some terms to help us. If I want to talk about some language \( x \), then I will refer to \( x \) as the target language. The language that I will be describing \( x \) in, I will call the Meta-language. So if I want to talk about OCaml, then OCaml will be the target language, and English will be the Meta-language.

### 11.3 Correctness

When we talk about correctness, we basically mean, does the program run how we expect it to run? Can I prove that \( + 2 3 \) returns 5 in Math-ew? How can I prove that \( (+ 2 (* 3 4)) \) returns 14 in LISP? The answer to this is not much different than proving that \( ((p \land q) \land (p \Rightarrow r) \land (q \Rightarrow r)) \Rightarrow r \). Namely, we can make a proof:

\[
\begin{align*}
 p \land q \\
p \Rightarrow r \\
q \Rightarrow r
\end{align*}
\]

\[
\therefore r
\]

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 \( 2+3 \) is \( 14 \) or \( (+ 2 ( \ast 3 4)) \) returns 14. However instead of using defined rules of algebra or logic that we know, we are going to use defined rules of the target language.

### 11.4 Operational Semantics

Let us define a very basic language \( A language: \)

\[
\begin{align*}
e & \rightarrow n \\
\rightarrow e ? e \\
n & \rightarrow 0|1|2|3|...
\end{align*}
\]

\( A \) 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:

\[
n \Rightarrow n
\]

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 \( ?4 \) 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 \( e ? e \).

\[
e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n_3 \text{ is } n_1 + n_2 \\
e_1 ? e_2 \Rightarrow n_3
\]

This is to say that \( e_1 \) and \( e_2 \) are some expressions, and that \( e_1 \) will eventually evaluate to some number, \( n_1 \), while \( e_2 \) will eventually evaluate to some number \( n_2 \). We then need to describe that we want to add the two numbers together to get a final value: \( n_3 \) is \( n_1 + n_2 \). This part is described in our meta language. We then want to show that if these statements are true, then when we see \( e_1 \ ? e_2 \) that we want to return \( n_3 \), whatever that is.

This particular example that uses \( ? \) instead of \( + \), is just to show you that we can just arbitrarily use symbols to stand for symbols and as long as we describe what this symbol does in our target language, then we can show you a rule of what is supposed to happen.

Now that we have our two rules to match each thing in our grammar, we can start making proofs that show what would happen if we had a statement like \( 4 ? 3 \).

In this example, looking at our grammar, we can see that 4 is \( e_1 \) and that 3 is \( e_2 \). We also know that, 4 and 3 are just numbers which we know evaluate to themselves. So constructing a proof of correctness for the statement \( 4 ? 4 \) using the above two rules would look like:

\[
\begin{align*}
4 \Rightarrow 4 \\
3 \Rightarrow 3
\end{align*}
\]

\[
7 \text{ is } 4 + 3 \\
4 ? 3 \Rightarrow 7
\]
That also means that we could prove larger expressions such as 3?4?5. Here I will assign 3 to \( e_1 \) and 4?5 to \( e_2 \). 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:

\[
\begin{align*}
3 \Rightarrow 3 & \quad 4 \Rightarrow 4 \quad 5 \Rightarrow 5 \\
3 + 4 \Rightarrow 9 & \quad 9 \Rightarrow 4+5 \\
3 \cdot 4 \Rightarrow 12 & \quad 12 \Rightarrow 3 + 9
\end{align*}
\]

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

\[
\begin{align*}
e & \rightarrow n \\
& \rightarrow e + e \\
& \rightarrow V \\
& \rightarrow \text{let } V = e \text{ in } e \\
n & \rightarrow 0|1|2|3|\ldots \\
V & \Rightarrow a|b|c|d|\ldots
\end{align*}
\]

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:

\[
\begin{align*}
A; n & \Rightarrow n \\
A; e_1 \Rightarrow n_1 & \quad A; e_2 \Rightarrow n_2 \\
& \quad n_3 \Rightarrow n_1 + n_2 \\
A; e_1 ? e_2 & \Rightarrow n_3
\end{align*}
\]

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:

\[
A, x : 4; 6 \Rightarrow 6
\]

I still include \( A \) because there are probably other environment variables that we are unaware of.

However, this is quite a boring example. What we may care about is how to look up a variable in our language. That is, what is the rule for \( e \rightarrow V \)? If we want to evaluate \( V \) into a value, we need to look up that value in the environment. Thus, our rule has to describe this process. Conventionally, we do this in the following way:

\[
\begin{align*}
A(x) & \Rightarrow v \\
A; x \Rightarrow v
\end{align*}
\]

So if had previously bound the variable \( x \) to the value 4 and wanted to look up \( x \), it would look like:

\[
A, x : 4; (x) \Rightarrow 4
\]

This rule looks like it’s just a repetition of a line, but recall the idea of the target language and the meta language. The conclusion is describing the target language, while the premise or hypothesis is describing what to do in the meta language.

We did do this a bit out of order. Before we can look up anything, we would have first needed to bind something. So let’s describe the rule of \( e \rightarrow \text{let } V = e_1 \text{ in } e_2 \).

In this case, following OCaml (this is not always the case), before we bind a value to a variable, we want to evaluate the expression \( e_1 \) to a value and then bind that resulting value to the variable. Then we want to use this new binding when we are evaluating the expression \( e_2 \). Consider \( \text{let } x = 3 \text{ in } x + 1 \). \( x+1 \) 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:

\[
A; e_1 \Rightarrow v \\
A, x : v ; e_2 \Rightarrow e_3
\]

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 \( x + 4 \).

\[
A; e_1 \Rightarrow v \quad A, x : v; e_2 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

In this case we identify that 3 is \( e_1 \) and \( x + 4 \) is \( e_2 \).

\[
A; 3 \Rightarrow v \quad A, x : v; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

We know that \( x = 3 \) so

\[
A; x : 3; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

We then want to use our plus rule when evaluating \( x + 4 \)

\[
A\Rightarrow 3 \quad A, x : 3; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

Here we can do what we did above and notice that in \( x + 4 \) that \( x \) is \( e_4 \) and 4 is \( e_5 \).

\[
A\Rightarrow 3 \quad A, x : 3; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

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

\[
A\Rightarrow 3 \quad A, x : 3; x \Rightarrow 3 \quad A, x : 3; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

We know that 4 evaluates to itself:

\[
A\Rightarrow 3 \quad A, x : 3; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

and we know that \( 3 + 4 \) is the value of 7:

\[
A, x : 3; (x) \Rightarrow 3 \\
A, x : 3; x + 4 \Rightarrow 4 \\
\text{7 is } 3 + 4
\]

\[
A\Rightarrow 3 \quad A, x : 3; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3
\]

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

\[
A\Rightarrow 3 \quad A, x : 3; x + 4 \Rightarrow e_3 \\
\text{let } x = 3 \text{ in } x + 4 \Rightarrow 7
\]

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 \( A, x : v; e \) 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 \textit{C Language} (not to be confused with C):

\[
e \Rightarrow n \\
e \Rightarrow e + e \\
e \Rightarrow V \\
e \Rightarrow \text{let } V = e \text{ in } e \\
e \Rightarrow B \\
e \Rightarrow \text{if } e \text{ then } e \text{ else } e \\
n \Rightarrow 0|1|2|3|\ldots \\
V \Rightarrow a|b|c|d|\ldots B \Rightarrow \text{true}|false
\]

We should add some rules to incorporate these new values.

\textit{TODO, but enough to post notes}
11.5 Definitions interpreter

Let us go back to our very simple Language. 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 Language

\[
\begin{align*}
A; n & \Rightarrow n \\
A; e_1 & \Rightarrow n_1 & A; e_2 & \Rightarrow n_2 & n_3 \text{ is } n_1 + n_2 \\
A; e_1 + e_2 & \Rightarrow n_3
\end{align*}
\]

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:

```ocaml
let 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 Language, we gain more rules:

\[
\begin{align*}
A(x) & \Rightarrow v \\
A; x & \Rightarrow v \\
A; e_1 & \Rightarrow v & A; x; e_2 & \Rightarrow e_3
\end{align*}
\]

We can then update our interpreter accordingly:

```ocaml
let 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 a function that adds variable and value pairs to the environment and a function that looks up a variable in the environment.
Chapter 12

Lambda Calculus

12.1 Intro

According to Wikipedia¹, "Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution."

While correct, we will be treating it as a Turing complete language which is the basis of many functional languages. So we will go over how to create statements in this language, and how to evaluate them.

Let us begin by giving the grammar for the language:

\[
e \Rightarrow x \\
| \lambda x.e \\
| ee
\]

\(x\) here is a variable. That's it. A very small language grammar. Using this grammar the following statements are valid:

\[\begin{align*}
  x & \quad x \\
  y & \quad \lambda x.x \\
  \lambda x.y & \\
  \lambda y & \\
  \lambda x.\lambda y.x & \\
  \lambda x.x & \\
  \lambda x.\lambda y.x & \\
  \lambda x.y & \\
  \lambda x.xy & \\
  \lambda x.\lambda y.x & \\
  \lambda x.y & \\
  \lambda x.xy & \\
\end{align*}\]

¹https://en.wikipedia.org/wiki/Lambda_calculus